ILLUSTRATING FORTRAN

(THE PORTABLE VARIETY)

DONALD ALCOCK

ILLUSTRATING FORTRAN

ILLUSTRATING FORTRAN

(THE PORTABLE VARIETY)

DONALD ALCOCK

CAMBRIDGE UNIVERSITY PRESS

CAMBRIDGE
LONDON NEW YORK NEW ROCHELLE
MELBOURNE SYDNEY

Published by the Press Syndicate of the University of Cambridge,
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
296 Beaconsfield Parade, Middle Park, Melbourne 3206, Australia

First published 1982

Printed in Great Britain at the University Press, Cambridge

British Library Cataloguing in Publication Data

Alcock, Donald
Illustrating Fortran

1. FORTRAN (Computer program language)
2. Electronic digital computers – Programming
I. Title
001.64'24 QA76.73.F25

ISBN 0 521 24598 2 hard covers
ISBN 0 521 28810 X paperback

Acknowledgements

I am grateful to Cambridge University Press for permission to model certain material and examples in this book on my earlier book *Illustrating BASIC*. This includes the bug ; computer jargon for a mistake in a program. My thanks, also, to Brian Shearing for permission to re-use ideas and some material from the *MP/3 Fortran Reference Manual* published by *Euro Computer Systems Limited*.

I am humbly grateful to Colin Day for patiently typing the examples in this book and proving them on a computer — thereby preventing some very silly bugs being published.

This book would not have been finished without love and encouragement from family and friends when I felt too depressed to carry on. My warm thanks, here, to Peter Golden, the Chessers, Bonner Mitchell, John Shaw and Ken Peek.

CONTENTS

PREFACE

Fortran (*FORmula TRANslation*) is a computer language that has been around for a quarter of a century: publication of this book coincides with the twenty-fifth anniversary of the first Fortran compiler. Fortran is still much used despite predictions throughout its life that more elegant languages would supersede it. But Fortran is still the only language in which it is possible — with care — to write really *portable* programs. A portable program is one that should work with little or no alteration on any computer that has Fortran: in other words on most computers of any power. Some firms employing less than twenty people have invested more than fifteen man-years in Fortran programs: in big companies such investment can be enormous. The potential portability of Fortran has enabled such firms to transfer their investment from an outdated computer to a modern one — in some cases several times during the firm's history. Changing to a more elegant programming language would be more expensive than sticking to Fortran — so Fortran is here to stay for a while.

The Fortran described and illustrated in this book is that defined by the American National Standards Institute in 1966 (Fortran 66; roughly Fortran IV) but with allusions to the same Institute's standard published in 1978 and affectionately known as Fortran 77. At present not every computer can understand the new language, but even if it *were* possible to write widely portable programs in full Fortran 77 it would not be practical to present the whole language in a book such as this. Fortran 77 — of which Fortran 66 is a subset — is a **BIG** language.

THICKNESS OF ANSI REPORT: FORTRAN 66

THICKNESS OF ANSI REPORT: FORTRAN 77

The most important aim of this book is to provide a reference manual for Fortran emphasizing the self discipline needed to achieve portable programs. In deciding what could safely be advocated for portability I referred to *Compatible Fortran* and the *HECB Programming Instruction Manual* — both cited in the bibliography. Apart from using *INTEGER* and *REAL* declarations (forbidden by the *HECB* manual) I believe my presentation of Fortran 66 takes account of all other important recommendations made in those two books. I have described every facility in Fortran 66 but encouraged the reader not to use (or use only with caution) those that might lead to non-portable programs.

The second aim of this book is to introduce programming to the person who has not tried to program a computer before, or has used only the ubiquitous language called *BASIC*. No previous knowledge of computers or computing is assumed, but the novice may not be able to take the advice given to Alice: "Begin at the beginning ... and go on till you reach the end: then stop." That is because this book, being in the form of a reference manual, has forward references here and there. So the novice

is invited to skip certain pages (these are few and clearly marked) until he or she has a rough grasp of programming in Fortran before returning to cope with the subtleties.

The third aim of this book is to illustrate a few tricks of the programmer's trade such as sorting numbers, plotting graphs, solving simultaneous equations, finding routes through networks and decoding algebraic expressions. Several of the examples show how to handle characters (letters, digits and symbols) in programs that should nevertheless be portable.

The examples of Fortran programs in this book are written with the idea of clarity in mind; not efficiency. A program is seldom a static thing; more often it remains in a state of periodic improvement, extension and correction. An intelligible program is easier to maintain than one that sacrifices clarity for efficiency. Because programmers' time gets ever more expensive – and electronic computation cheaper and cheaper – the clearly written program makes more sense economically than the "clever" and efficient one. Even so the reader is sure to find pieces of program in this book that cry out for greater efficiency with little loss of clarity. By all means note any desired alterations in the margins of this book wherever my programming offends your sense of style or thirst for efficiency.

In claiming the advantages of clarity some may protest there are few *comment* lines in my examples. That is because most comments are made in adjacent explanatory texts or in little clouds like this so as to fit a complete example on one page.

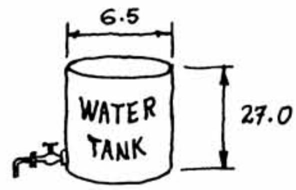
Reigate,
Surrey, U.K.

Donald Alcock
May 1982

1

INTRODUCTION

THE CONCEPT OF A "PROGRAM" WITHOUT INVOLVING A COMPUTER



Assume there is no computer to help solve this problem confronting a painter: how many pots of paint does he need to paint the roof and wall of this water tank ?

The paint manufacturer says each pot has enough paint to cover an area of 236.

Remembering that the area of a circle is given by πr^2 (where r is its radius) or $\pi d^2 \div 4$ (where d is its diameter) the painter can work out the area of the top of the tank:

$$\text{AREA OF TOP} = 3.14 \times 6.5^2 \div 4 = 33.1663$$

Remembering that the circumference of a circle is given by πd (where d is its diameter as before) the painter can work out the area of the wall of the tank:

$$\begin{aligned} \text{AREA OF WALL} &= \text{CIRCUMFERENCE} \times \text{HEIGHT} \\ &= 3.14 \times 6.5 \times 27.0 = 551.070 \end{aligned}$$

The area to be painted is the sum of the areas of top and wall:

$$\text{TOTAL AREA} = 33.1663 + 551.070 = 584.2363$$

Into this area must be divided the coverage of each pot of paint so as to give the number of pots needed:

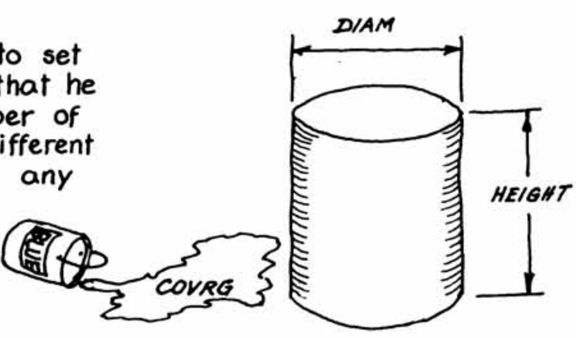
$$\text{POTS} = 584.2363 \div 236.0 = 2.47558$$

You cannot buy a fraction of a pot of paint, so the final answer must be rounded up to the next whole number. (Looking at it another way, take the *integral part* of the value and add unity.)

$$\text{NUMBER OF POTS} = 2\cancel{.47558} + 1 = 3 \leftarrow \text{THE SOLUTION}$$

If the items of data had been such that POTS had worked out at, say, 3.00000 instead of 2.47558 then the solution would have been 3 + 1 = 4 pots. Although this would be wrong *mathematically* a prudent painter would be happier with the extra pot in case he spilled a few drops of paint.

NOW suppose the painter wanted to set down this method of calculation so that he could more quickly calculate the number of pots of paint (having perhaps a different capacity) to paint a water tank of any diameter and any height.



A possible list of instructions is set out on the opposite page.

A. Draw three little boxes to contain the items of data needed to solve the problem. Name these boxes *DIAM*, *HEIGHT*, *COVRG*:

DIAM [ ] *HEIGHT* [ ] *COVRG* [ ]

also draw boxes to contain intermediate results of the calculation:

TOP [ ] *WALL* [ ] *POTS* [ ]

B. Draw a little box to contain the number of pots needed; but draw it differently to emphasize that its contents must be a whole number ≈ an *integer*. *NPOTS* [ ]

C. Read three items of data for a particular case to be solved, and put them in the boxes named *DIAM*, *HEIGHT*, *COVRG*.

≈ *EXAMPLE* ≈

DIAM [6.5] *HEIGHT* [27.0] *COVRG* [236.0]

D. Work out the area of the top of the tank by the formula $3.14 \times d^2 \div 4$ where d is the diameter found in the box named *DIAM*. Put the answer in the box named *TOP*.

TOP [33.1663] ← $3.14 \times 6.5^2 \div 4.0$ ≈ *EXAMPLE* ≈

E. Work out the area of the vertical wall of the tank by the formula $3.14 \times d \times h$ where d and h are the diameter and height found in the boxes named *DIAM* and *HEIGHT* respectively. Put the answer in the box named *WALL*.

≈ *EXAMPLE* ≈ *WALL* [551.070] ← $3.14 \times 6.5 \times 27.0$

F. Add the areas found in boxes named *TOP* and *WALL*. Divide this sum by the coverage of a pot of paint. This is found in the box named *COVRG*. Put the answer in the box named *POTS*.

≈ *EXAMPLE* ≈ $(33.1663 + 551.070) \div 236.0$ → *POTS* [2.47558]

G. Now take the integral part of the number found in the box named *POTS*; add 1; put the result in the *integer* box named *NPOTS*.

≈ *EXAMPLE* ≈ 2(.47558) +1 → *NPOTS* [3]

H. Write out the solution; in other words write out the integer found in the box named *NPOTS*. Give this note to the painter.

≈ *EXAMPLE* ≈

you need 3 pots of paint

I. STOP work: there is nothing more to do.

J. This is the *END* of the list of instructions.

In computer jargon such a list of instructions (excluding the embedded examples) is called a *program*; the person who writes programs is called a *programmer*. The above program is written in English, but over the page it is translated into *Fortran* which is a language a *computer* can understand and obey. (The jargon for *obey* is *execute*.) The instructions are obeyed in sequence starting at the first and stopping on meeting *STOP*.

ILLUSTRATION OF A FORTRAN PROGRAM TO SOLVE THE WATER TANK PROBLEM

```
      REAL DIAM, HEIGHT, CØVRG, TØP, WALL, PØTS          A
      INTEGER NPØTS                                      B
      READ (5,100) DIAM, HEIGHT, CØVRG                   C
  100 FØRMAT (3F10.0)
      TØP = 3.14 * (DIAM**2) / 4.0                       D
      WALL = 3.14 * DIAM * HEIGHT                        E
      PØTS = ( TØP + WALL ) / CØVRG                      F
      NPØTS = INT(PØTS) + 1                              G
      WRITE (6,200) NPØTS                                H
  200 FØRMAT (1X, 9H YØU NEED, I2, 5H PØTS )
      STØP                                               I
      END                                                J
```

Here is the program in *Fortran*. The letters *A* to *J* with their little arrows (*not* part of the Fortran) correspond to steps in the English program on the previous page. There are many niceties embodied in the Fortran notation above so it worth working right through the program yet again — pretending to *be* the computer.

Keep the items of data the same as before so as to simplify comparison with the program written in English. These items are written on a special form rather like the one used for the Fortran program above. Character positions are called *columns* for reasons explained later.

```
      6.5      27.0     236.0
         ^         ^         ^         ^
     COLUMN    COLUMN    COLUMN    COLUMN
       10        20        30        40
```

Start obeying the Fortran program at its first *instruction* (also called a *statement*) and continue in sequence until you meet one which says *STOP*.

```
      REAL DIAM, HEIGHT, CØVRG, TØP, WALL, PØTS          A
```

This tells the computer to name six "little boxes" with the names shown. For the English program we *drew* the boxes. The computer, however, uses storage locations each capable of storing a *REAL* number (in other words a number with a fractional part after a decimal point). The locations *DIAM*, *HEIGHT*, *COVRG*, *TOP*, *WALL*, *POTS* are called *REAL variables*.

```
      INTEGER NPØTS                                      B
```

This tells the computer to name another "little box" *NPOTS*. In the English program we drew this box differently from the boxes designed to hold *REAL* numbers. This box is designed to hold an *INTEGER*. The location *NPOTS* is called an *INTEGER variable*.

```
      READ (5,100) DIAM, HEIGHT, CØVRG                   C
  100 FØRMAT (3F10.0)
```

This instructs the computer to read some waiting items of data and put them into variables *DIAM*, *HEIGHT*, *COVRG*. The *FORMAT* statement says there should be 3 waiting items, each in *fixed point* form (hence the *3F*) and that they reside in successive *fields* of 10 columns each (hence the *10.0*). Chapter 10 deals with formats in detail; until then all examples use *F10.0* for reading real numbers which should be written with decimal points.

The *100* (in *READ(5,100)*) associates the *READ* statement with its *FORMAT* statement. This number is arbitrary and the two statements do not have to follow one another. Indeed it is common practice to group *FORMATs* together.

The 5 (in *READ(5,100)*) is a *unit number*. You can read from all sorts of peripheral devices each of which is associated with an unique unit number. Conventionally unit 5 denotes a *card reader*. All examples until Chapter 11 have 5 as the unit number in *READ* statements.

```
      TØP = 3.14 * (DIAM**2) / 4.0
```
D

This is as described for step D of the English program, but notice the use of * for *multiply by*, the use of ** for *raise to a power*, the use of / for *divide by*. The brackets are not necessary here (exponentiation is always done first) but they clarify the instruction and do no harm.

For the equals sign don't say "equals" (this is not an equation in algebra) say "becomes". Thus: *TOP becomes 3.14 times DIAM to the power 2 divided by 4.0*. Statements with an equals sign are called *assignment statements* or *assignments*.

```
      WALL = 3.14 * DIAM * HEIGHT
```
E

This assignment says: *WALL becomes 3.14 times DIAM times HEIGHT*. It corresponds precisely to step E of the English program.

```
      PØTS = ( TØP + WALL ) / CØVRG
```
F

This assignment says: *POTS becomes (TOP plus WALL) all divided by COVRG*. Notice the importance of the brackets. Without them *POTS* would become *TOP* plus *WALL/COVRG* which is ridiculous. In the absence of brackets, multiplications and divisions are done before additions and subtractions ~ just as you would expect from the rules of algebra.

```
      NPØTS = INT(PØTS) + 1
```
G

This assignment has a *function* on the right-hand side. This function, *INT( )*, causes the computer to consult whatever real number is represented inside the brackets and take just its integral part. Thus the statement says: *NPOTS becomes the integral part of POTS plus one*. (Fortran has many other functions as described in Chapter 6.)

```
      WRITE (6,200) NPØTS
  200 FØRMAT (1X, 9H YØU NEED, I2, 5H PØTS )
```
H

This instructs the computer to write out the value found in the integer variable *NPOTS*. The *1X* in the *FORMAT* statement says to start a new line. The *9H* says to print the following 9 characters " *YOU NEED*" which includes the space before *YOU* and the space before *NEED*. The *I2* says to print the integer (found in variable *NPOTS*) in a field of two columns, justified to the right. The *5H* says to print the following 5 characters " *POTS*" including the initial space. So the result should look like this:

```
YOU NEED 3 POTS
```

Why does *H* signify *characters* ? This is history. Hermann Hollerith pioneered punched-card equipment in the 1890s and his initial, H, survives in the terminology of Fortran. Characters may be called *Hollerith characters*.

The 200 (in *WRITE(6,200)*) associates the *WRITE* with the *FORMAT* statement as explained for the *READ* statement above. Again this number is arbitrary and the *FORMAT* statement need not follow the *WRITE*. The 6 denotes a *unit number* as described for the *READ* statement. Conventionally *unit 6* is associated with a *line printer* and is so used throughout the examples in this book.

```
      STØP
```
I

This tells the computer to stop obeying instructions ~ in other words to stop execution.

```
      END
```
J

This simply marks the end of a deck of cards for a program like a lid on a box. It is *not* an instruction: it is called *the END line*.

PREPARATION

GETTING A PROGRAM AND ITS DATA INTO A FORM A COMPUTER CAN READ

Here is a program and its data written on standard coding forms:

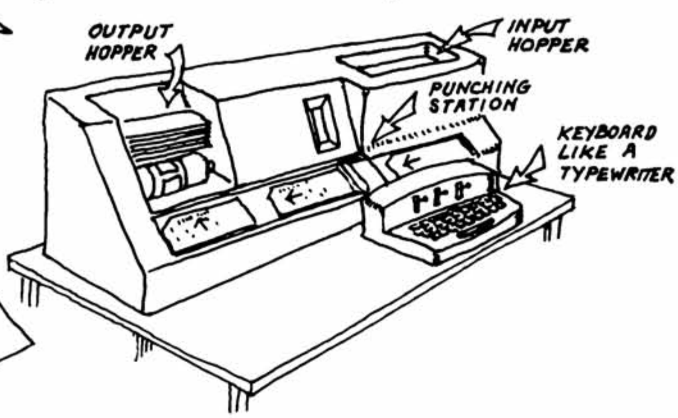
FORTRAN PROGRAM name: author: date:

```
      REAL DIAM, HEIGHT, CØVRG, TØP, WALL, P
      INTEGER NPØTS
      READ (5,100) DIAM, HEIGHT, CØVRG
  100 FØRMAT (3F10.0)
      TØP = 3.14 * (DIAM**2) / 4.0
      WALL = 3.14 * DIAM * HEIGHT
      PØTS = ( TØP + WALL ) / CØVRG
      NPØTS = INT(PØTS) + 1
      WRITE (6,200) NPØTS
  200 FØRMAT (1X, 9H YØU NEED, I2, 5H PØTS)
      STØP
      END
```

FORTRAN DATA program: user: date:

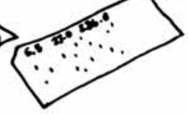
```
       6.5      27.0     236.0
```

The *traditional* way of encoding such material is to punch cards at a device called a *keypunch*.

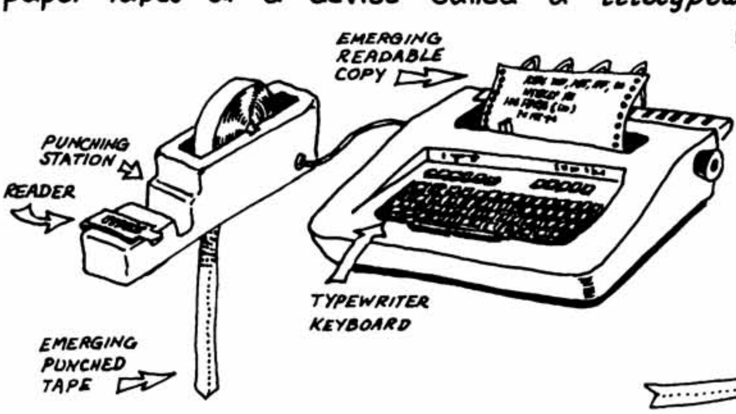


This leaves you with *two* decks of cards: a *program* deck

and a *data* deck.



Another traditional way of encoding programs and data is to punch paper tapes at a device called a *teletypewriter*.

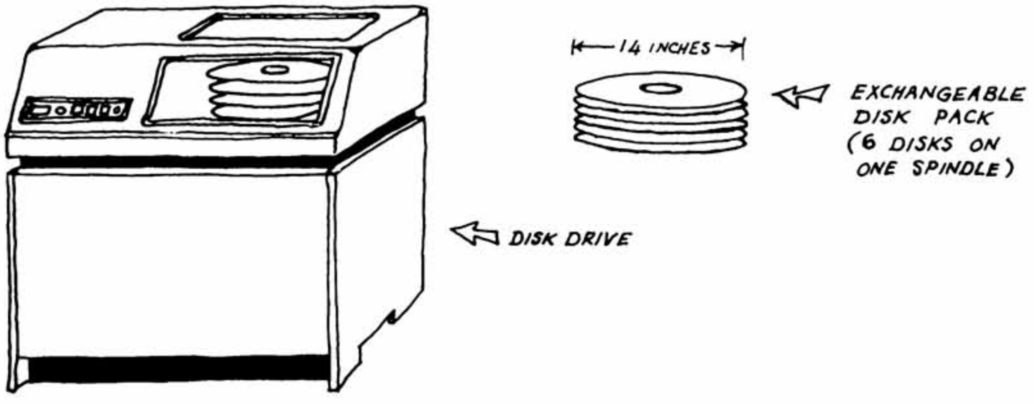


This leaves you with two reels of punched paper tape: a *program* tape,

and a *data* tape.

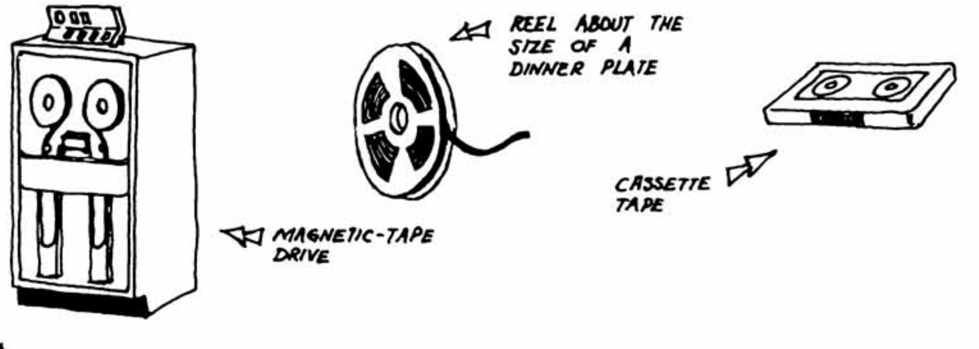
Nowadays the most common method of getting a program and data into a computer is to type the material at a *visual display unit* (VDU for short).

The information from this source is stored magnetically on the surface of a disk. The program constitutes a *Fortran file* and the data constitutes a *data file*. The disk may be a "hard" disk:



or a "floppy" disk:

or the information may be stored as files on a reel of magnetic tape or on a *cassette tape*:



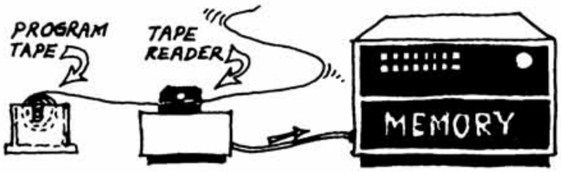
Whatever the medium there are ways of correcting mistakes in typing; ways of deleting lines, inserting lines, changing individual characters within lines, and so on. This is called *editing*. And you end up with two sets of information the computer can read:

- a program
- a set of data

EXECUTION

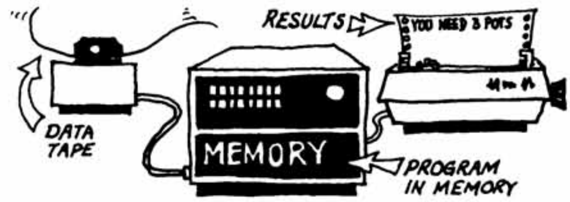
WHAT THE COMPUTER HAS TO DO WITH YOUR PROGRAM AND DATA

Using the "traditional" forms of input for illustration, what *appears* to happen when you feed your program and data to the computer is this:



You put the program tape in the tape reader (or program deck into the card reader) and the computer reads your program into its memory.

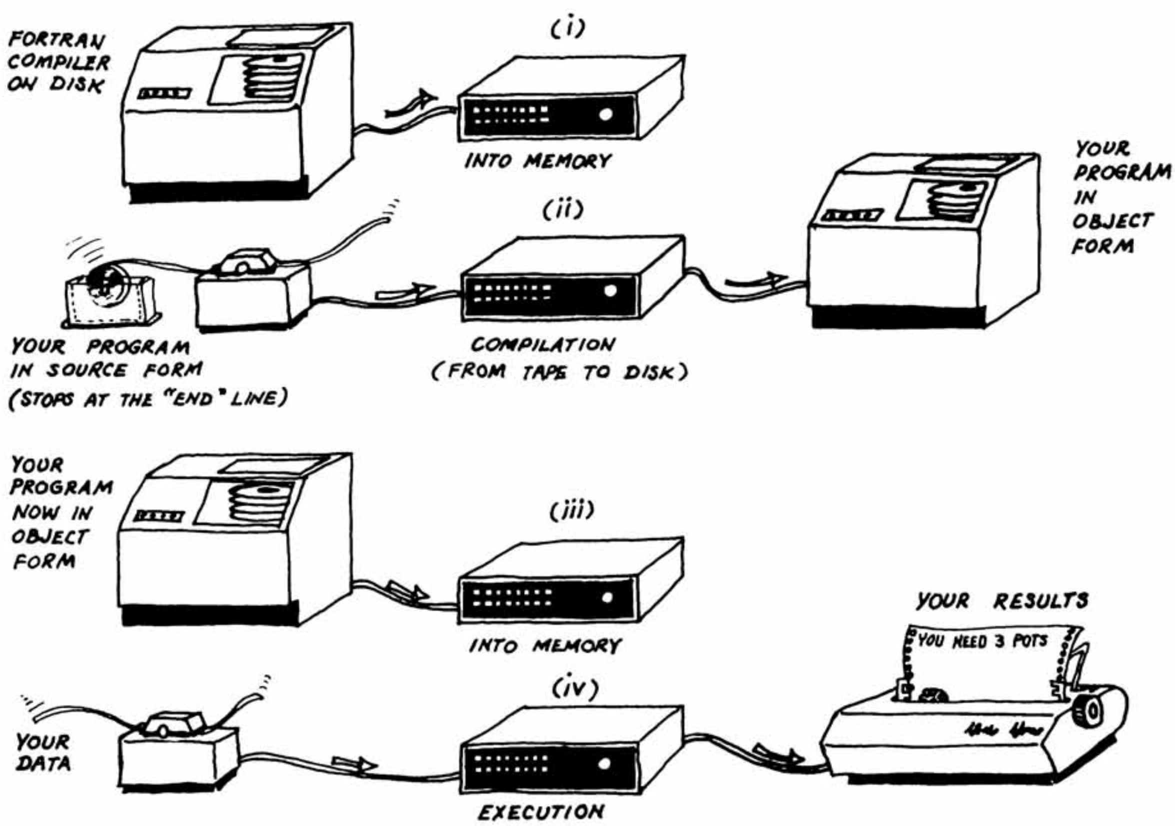
Then you put the data tape into the tape reader (or data deck into the card reader) and the computer starts to execute your program — taking its data from the reader and printing results on the line printer.



But it doesn't happen as simply as that. The computer can certainly understand and execute instructions, but only a program in *binary* form — something like this:

0010111001011010
0010100101001011
1010010110101101
1110010110000110
etc. etc.

One such program is called a *Fortran compiler*. This program is able to read *your* Fortran program and translate it into 0s and 1s — then hand control over to the first instruction of your newly-translated program. In computer jargon a compiler *compiles* a program from *source* form (for example from Fortran form) to *object* form (0s and 1s). So the simple two-stage process depicted above is more nearly as follows:



OPERATING SYSTEMS

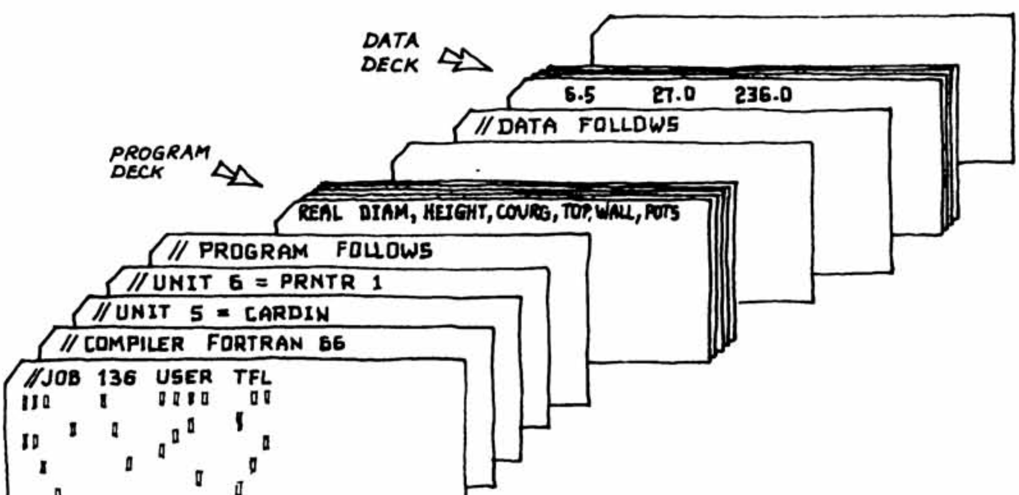
TELLING THE COMPUTER WHAT RESOURCES TO USE ~ AND WHEN

The computer has to be told in detail what compiler to use and what *peripheral devices* (card reader, tape reader, magnetic tape drives, disk files, card punch, tape punch, line printer) to use. This cannot be done just by throwing switches or pressing buttons. You have to instruct an *operating system*.

An operating system is a clever program rather like a compiler. Its job is to make the connections referred to opposite ~ for example ensuring the Fortran compiler on disk is read into the computer's memory, or that results from a particular program are routed to the line printer. Just as you have to know the language called *Fortran* to write the kind of instructions illustrated on page 6, you *also* have to know the language of the operating system to tell the computer what peripheral devices to use and what compiler to employ.

The language of an operating system is called a *job-control language* (*JCL* for short). Every computer has one. Even a modest microcomputer has a JCL ~ with a vocabulary perhaps no more extensive than the words *LOAD* and *SAVE* and *RUN*. But a large modern computer serves many users simultaneously ~ each wanting to employ a different compiler (Fortran, COBOL, Pascal, APL *etc.*) or be competing for the next turn at one of the line printers. Such an installation has a daunting JCL with huge vocabulary and complicated grammar. You may have to "talk" to such an operating system over a telephone line by typing *commands* in JCL at a visual display unit.

Below is illustrated a "traditional" program deck and data deck ~ interspersed with commands to an operating system written in a typical (but imaginary) JCL.



This book does not describe operating systems because there are so many of them and they are so different in nature from one make of computer to another. There are few short cuts to learning the JCL of an operating system. You should persistently ask questions of those who seem to know it all ~ study whatever manual is available even if it seems, at first, incomprehensible ~ and above all try out your efforts fearlessly. If the operating system then makes a mess of everything in the computer, losing everyone's files, it is **NOT YOUR FAULT.**

EXERCISES

CHAPTER 1

1.1 Try making the program about water tanks run on the computer to which you have access. Try it with several different sets of data. (This exercise might take longer than you expect.)

1.2 Note the necessary job-control cards (or what you have to type when "signing on" and submitting such a job from your terminal) for future reference:

JCL procedure for the .... installation

2

STRUCTURE

PUNCHED CARDS
LINES
LABELS
PROGRAM UNITS *
ORDER *
EXERCISES

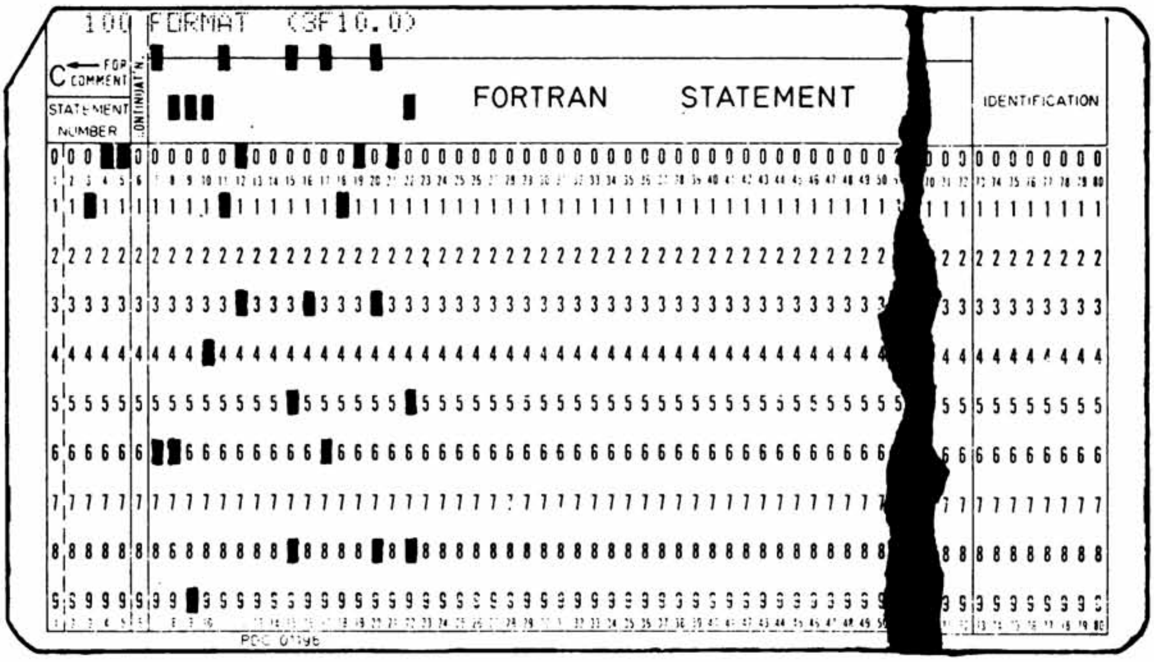
* SKIP ON FIRST READING

PUNCHED CARDS

HERMANN HOLLERITH PROVIDING SOME OF THE TERMINOLOGY OF FORTRAN

You may not have to use punched cards. Nowadays it is common to use Fortran from a terminal to a large computer ≈ or on a personal computer where the means of communication is a typewriter keyboard. But Fortran was originally designed with *punched cards* in mind ≈ and some ideas and terminology derived from punched cards remain in the terminology of Fortran.

So here is a punched card:

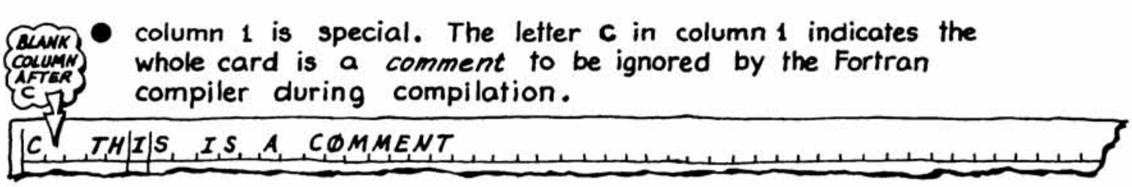


Each character occupies one *column* of which there are eighty per card.

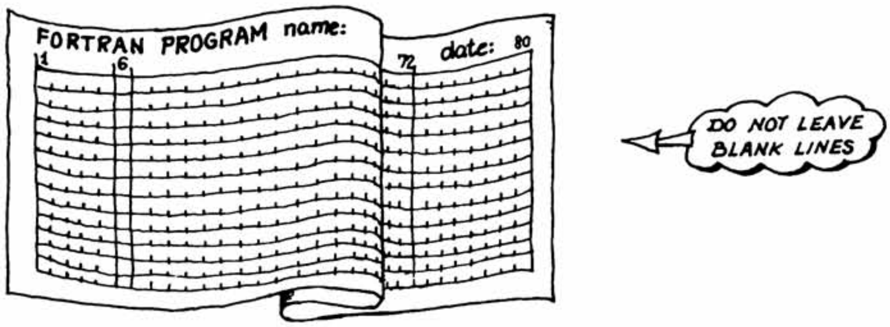
A card is punched by typing on a keyboard much like that of a typewriter where a touch on a key produces a corresponding pattern of holes in the column of a card. A touch on the space bar produces a blank column. A touch on a special key causes the current card to be ejected into the output hopper and a new card made ready for punching in column 1.

In Fortran statements the columns are employed as follows:

- blank cards are not allowed in the body of the program
- columns 1 to 5 are for the label if any (note labels 100 and 200 in the introductory example)
- column 6 is normally left blank: if not blank or zero this signifies continuation of a statement from the previous card
- columns 7 to 72 are for the body of the Fortran statement
- columns 73 to 80 are ignored by the Fortran compiler
- column 1 is special. The letter C in column 1 indicates the whole card is a *comment* to be ignored by the Fortran compiler during compilation.

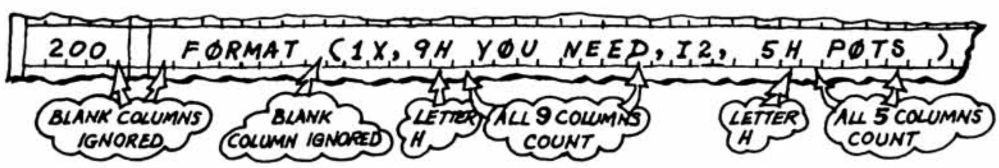


It is usual to write Fortran on *coding forms* divided into eighty columns and with columns 1, 6 and 72 emphasized in some way:

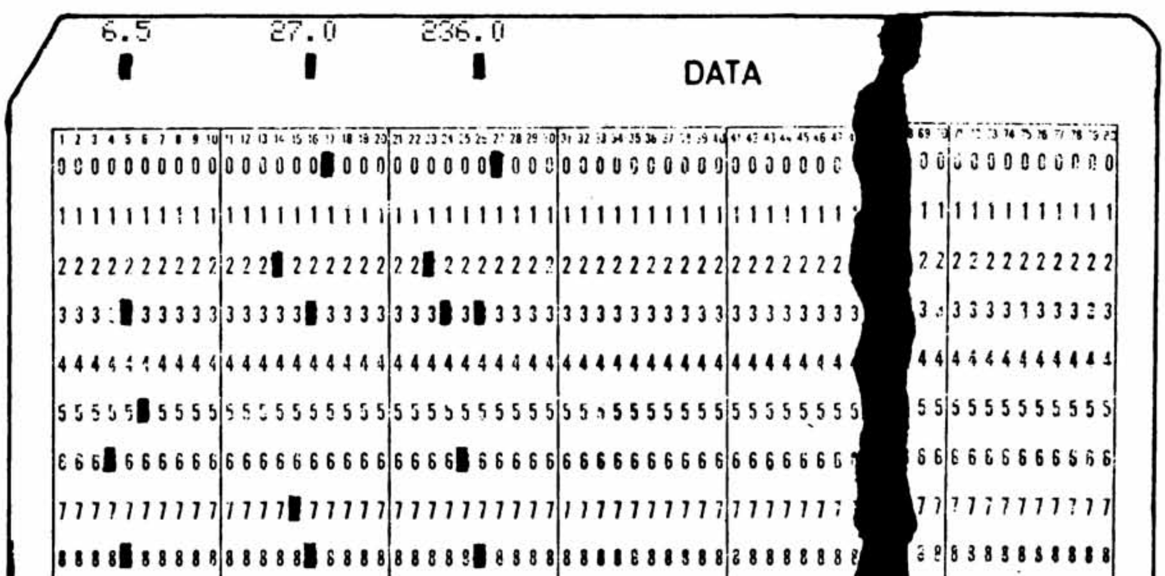


In the examples in this book column 6 is indicated by two vertical lines. Columns elsewhere in Fortran statements are usually not marked. Indication is unnecessary because blank columns are *generally ignored* inside Fortran statements in columns 7 to 72.

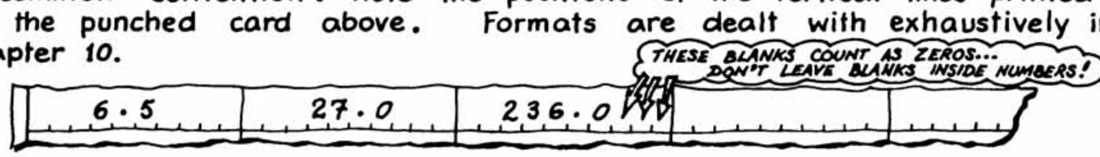
There is a notable exception to this. In Hollerith items blank columns are *not* ignored after letter *H*:



The rules for punching (or typing) *data* for a Fortran program are completely different from those set out above. So it is customary to employ a differently printed card.



Data are punched in *fields* as dictated by their respective *FORMAT* statements. In the introductory example *FORMAT(3F10.0)* demanded three numbers located in three successive fields of ten columns each. This is a common convention: note the positions of the vertical lines printed on the punched card above. Formats are dealt with exhaustively in Chapter 10.



LINES

THE CONTENT OF A SINGLE PUNCHED CARD ≈ OR ITS EQUIVALENT IN SOME OTHER MEDIUM

An *END line* is not like any other statement in Fortran. It simply marks the end of a program or subprogram (we meet subprograms over the page). The letters *E* then *N* then *D* are punched anywhere in columns 7 to 72 leaving the rest of the card blank:

```
|     |END                                                     |
```

NO LABEL   COLUMN 6   COLUMN 72

Some Fortrans (including Fortran 77) permit the *END* line to imply *STOP* in the main program and *RETURN* in a subprogram, but this is not allowed in Fortran 66.

A *continuation line* has a character other than blank or zero in column 6. There may be up to nineteen continuation lines following the *initial line* of a statement. The standard does not forbid it, but some Fortrans object to labels on continuation lines. Such labels can serve no useful purpose anyway.

```
      WALL = 3.14                  <- INITIAL LINE
     1           * DIAM            <- CONTINUATION LINES
     2             * HEIGHT
```

NO LABELS

The use of *1, 2 etc.* as the non-blank character in column 6 illustrates a tidy convention for continuation lines.

Never split a Hollerith item (such as *9H‸YOU‸NEED*) between one line and a continuation line. This would cause a Hollerith item to be filled out with blanks, thus making the next continuation line begin with rubbish. Although Fortran does not specifically forbid it, do not split any of Fortran's keywords (such as *FORMAT*) across continuation lines either; some Fortrans object to the practice.

```
  200 FØR
     1  MAT ( 1X, 9H YØU
     2                   NEED, I4, 5H PØ
     3TS)
```

EXTRA BLANKS COUNTED TO MAKE UP THE FULL 9 CHARACTERS

The *comment line* has letter C in column 1, then any comment (composed of characters from the standard set ≈ see page 20) in subsequent columns. It is safer to leave column 2 blank because there are Fortrans that attach some significance to characters here. Comment lines may be inserted anywhere before the *END* line except immediately before continuation lines.

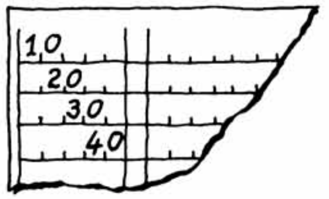
```
      REAL DIAM, HEIGHT, CØVRG, TØP
C THIS IS BEFØRE A CØNTINUATIØN LINE
     1WALL, PØTS
```

```
      STØP
C                              <- NO FOLLOWING TEXT ≈ THIS IS ALLOWED
      END
C THIS FØLLØWS AN END LINE
```

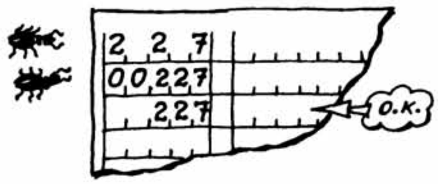
LABELS

FOR ANY EXECUTABLE STATEMENT AND ALL FORMAT STATEMENTS

A *label* may be written as digits anywhere in the first five columns:



Although standard Fortran permits, it is safer not to spread out the digits of a label or write a label with leading zeros:



Standard Fortran provides no limit to the size of integer used as a label – and different Fortrans impose different limits. A common limit (suggested as the one to ensure a portable program) is 32767 ($2^{15}-1$). Every label must be unique in its program or subprogram.

Statements in Fortran are either ***executable*** (*i.e.* cause something to happen when obeyed) or *non-executable* (*i.e.* declaratory in nature and not subject to being "obeyed"). Referring to the introductory example, statements are marked *EX* and *NONEX* respectively in the reproduction below:

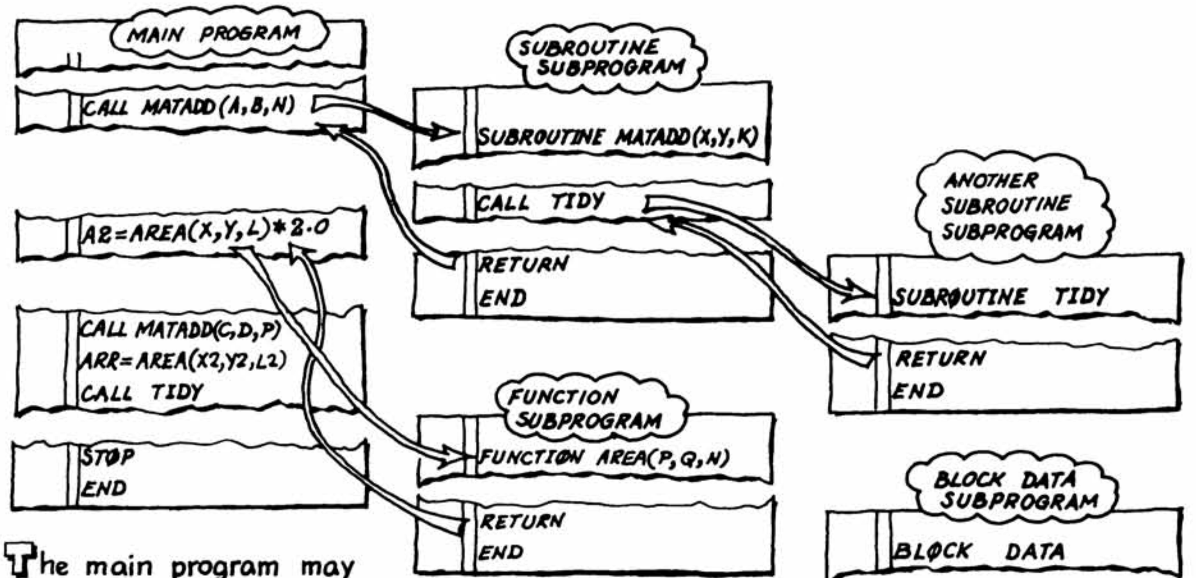
```
      REAL  DIAM, HEIGHT, CØVRG, TØP, WALL, PØTS         <- NONEX
      INTEGER  NPØTS                                     <- NONEX
      READ (5,100) DIAM, HEIGHT, CØVRG                   <- EX
100   FØRMAT  (3F10.0)                                   <- NONEX
      TØP = 3.14 * (DIAM **2) / 4.0                      <- EX
      WALL =  3.14 * DIAM * HEIGHT                       <- EX
      PØTS =  (TØP + WALL) / CØVRG                       <- EX
      NPØTS =  INT(PØTS) + 1                             <- EX
      WRITE (6,200) NPØTS                                <- EX
200   FØRMAT (1X, 9H YØU NEED, I2, 5H PØTS)              <- NONEX
      STØP                                               <- EX
      END                     <- NEITHER!
```

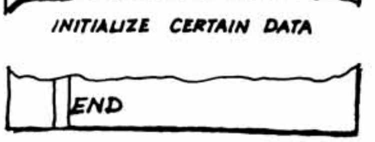
Any executable statement may be given a label. The only non-executable statement that may be labelled is the *FORMAT* statement – and that statement *must* be labelled to be of any use. (Standard Fortran does not specifically forbid labels on non-executable statements but these have been known to cause trouble in some otherwise standard Fortrans.)

Novices to programming should skip the rest of this chapter on first reading.

PROGRAM UNITS

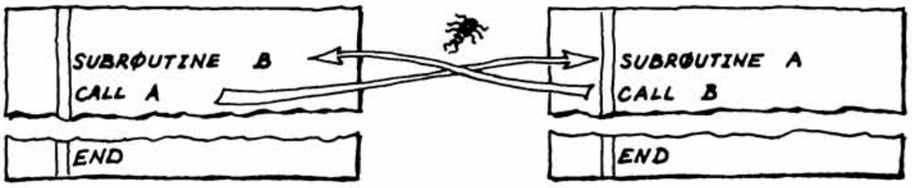
A Fortran program may be just a simple *main program* as illustrated in the introductory example. On the other hand the main program may refer to (or *invoke*) any number of *subprograms* any number of times.





The main program may share *COMMON* storage (see page 76) with one or more subprograms. Initial values may be set in such storage by a *BLOCK DATA* subprogram.

Subprograms invoked by the main program may, in turn, invoke other subprograms. The only restriction is that no main program or subprogram may invoke *itself* ⇔ either directly or indirectly. In other words *recursion** is not allowed.



There are three kinds of subprogram:

- *SUBROUTINE* (Chapter 7)
- *FUNCTION* (Chapter 7)
- *BLOCK DATA* (Chapter 9)

* it is possible to use the *principle* of recursion in Fortran by setting up your own stacks *etc.*. See Bibliography: *Fortran Techniques*.

Every subprogram begins with a heading containing one of the above names and ends with an *END line* as illustrated. The *main* program (of which there must always be precisely one) is distinguished by the absence of any special heading. Many Fortrans demand such a heading beginning with the word *PROGRAM* or *MAIN* but this should be considered part of the job-control language rather than a Fortran statement.

The term *program unit* is used to mean a main program *or* a subprogram.

ORDER

OF STATEMENTS IN A PROGRAM UNIT
(SKIP THIS PAGE ON FIRST READING)

The order of statements in a program unit is defined by the following table:

<table>
<tr><td colspan="3">One SUBROUTINE or FUNCTION or BLOCK DATA statement
(or nothing at all if this is the main program)</td></tr>
<tr><td rowspan="8">COMMENT lines
(anywhere between heading and END line but not between continuation lines)</td><td colspan="2">REAL, INTEGER, DOUBLE PRECISION, LOGICAL, COMPLEX
(i.e. type statements) in any order among themselves</td></tr>
<tr><td>DIMENSION statements</td><td rowspan="2">EXTERNAL statements
(intermingled among COMMON and DIMENSION statements)</td></tr>
<tr><td>COMMON statements</td></tr>
<tr><td colspan="2">EQUIVALENCE statements</td></tr>
<tr><td colspan="2">DATA statements (at least one in a BLOCK DATA subprogram)</td></tr>
<tr><td colspan="2">Statement functions</td></tr>
<tr><td>Executable statements
(at least one except in a BLOCK DATA subprogram)</td><td>FORMAT statements
(intermingled among the executable statements)</td></tr>
<tr><td colspan="3">END line (obligatory in every case)</td></tr>
</table>

This table (devised by Colin Day) prescribes an order more restrictive than that specified by Fortran 66, but an order more likely to result in a portable program. In Fortran 66 *FORMAT* statements are permitted the same freedom as *COMMENT lines*. *REAL, INTEGER, DOUBLE PRECISION, LOGICAL, COMPLEX, DIMENSION, COMMON, EQUIVALENCE* and *EXTERNAL* statements may be in any order among themselves.

EXERCISES

CHAPTER 2

2.1 Read what your Fortran manual says about *COMMENT lines*, *continuation lines*, labels, *END lines* and the allowable order of statements. Note in the margins of your manual where there should be tighter restrictions so as to achieve portable programs.

2.2 Note below your particular "house rules" for writing:

≈ letter *I*
digit *1*

≈ letter *O*
digit *0*

≈ letter Z

≈ digit 7

2.3 If you are using punched cards, punch one with all forty-seven characters of the standard set and stick the card in the space below (cut off the last thirty columns and it should fit). The standard character set is shown overleaf ≈ page 20.

3

ELEMENTS OF FORTRAN

CHARACTERS
SYMBOLIC NAMES
TYPES OF VARIABLE
TYPES OF CONSTANT
ARITHMETIC EXPRESSIONS
LOGICAL EXPRESSIONS *
ASSIGNMENT
LOANS (AN EXAMPLE)
EXERCISES

* SKIP ON FIRST READING

CHARACTERS

LETTERS, DIGITS & SYMBOLS

The Fortran 66 character set has forty-seven characters:

- the twenty six capital letters *A* to *Z*
- the ten digits *0* to *9*
- a *space* or *blank* (as made by pressing the space bar of a typewriter keyboard once)
- the ten special characters tabulated below:

=	*equals* (or *equals sign*)
+	*plus* (or *plus sign*)
-	*minus* (or *minus sign*)
*	*asterisk* (or *star*)
/	*slash* (or *oblique* or *solidus*)
(	*left parenthesis* (or *left bracket*)
)	*right parenthesis* (or *right bracket*)
,	*comma*
.	*decimal point* (or *point* or *full stop*)
$	*currency symbol* (or *dollar sign*) Although this is a standard character of Fortran there are some peripheral devices that cannot handle it ~ or print something different such as £

There is **NOTHING** implied by the order in which these characters are presented here.

Other characters are permitted in *COMMENT* lines and in Hollerith items but they are best avoided if a program is to be fully portable. Some peripheral devices are not capable of handling every non-standard character that might be employed.

```
  300  FØRMAT (8H%&abc#<>)
C AVØID  NØN-STANDARD CHARACTERS LIKE @[]↑!? etc
```

Fortran 77 adds the following two special characters:

'	*apostrophe* (or *quotation mark*)
:	*colon*

Lower-case (*i.e.* "small") letters are not standard characters in Fortran 66 or Fortran 77 ~ but there are many sophisticated programs for processing words and the texts of books. It would be foolish to refuse to use (or produce) such programs on the grounds that they have to be written in non-standard Fortran. But they would not, of course, be fully portable programs. If standard Fortran seems to be letting us down on this point, be assured that other languages demand similar restrictions. Fortran (with a few extensions) is often the only language available for writing programs to handle words and letters.

SYMBOLIC NAMES *NAMES DEVISED BY THE PROGRAMMER*

The introductory example began:

```
      REAL    DIAM, HEIGHT, CØVRG, TØP, WALL, PØTS
      INTEGER  NPØTS
```

and a little box was drawn for each variable:

DIAM	☐	HEIGHT	☐	COVRG	☐
TOP	☐	WALL	☐	POTS	☐
NPOTS	☐				

Each variable is given a *symbolic name* (or just *name*) by the programmer as illustrated above.

A symbolic name must begin with a letter and may contain anything from one to six letters and digits. (Some Fortrans permit more than six letters and digits but for the sake of portability keep the limit to six. The limit in Fortran 77 remains at six.)

Here are a few more symbolic names a programmer might invent:

A
N
H2SØ4
M46
RESULT

SKIP THE REST OF THIS PAGE ON FIRST READING

Symbolic names are for naming variables as already illustrated. They are also needed for naming:

- *Arrays* (Chapter 5)
- *Common blocks* (Chapter 8)
- *Statement functions* (Chapter 6)
- *Functions* (Chapters 6 & 7)
- *Subroutines* (Chapter 7)

Although Fortran permits one name to be used for more than one thing (for example, naming a common block X and naming a variable X) it is safer and less confusing not to do so. Likewise although there is no rule against using Fortran's own keywords as symbolic names (for example using the word STOP to name a variable) it is safer not to do this either. Two good rules are:

- keep symbolic names of variables, arrays and statement functions unique in each program unit
- keep the names of functions, subroutines and common blocks unique over the complete Fortran program

```
      IF ( STØP .EQ. IF+GØTØ) STØP
      GØTØ  10
```

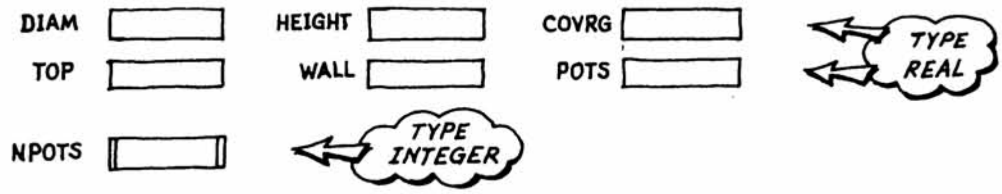
TYPES OF VARIABLE

REAL, INTEGER, DOUBLE PRECISION, LOGICAL, COMPLEX

The introductory example began:

```
REAL     DIAM, HEIGHT, CØVRG, TØP, WALL, PØTS
INTEGER  NPØTS
```

and a little box ~ a *variable* ~ was drawn for each item declared:

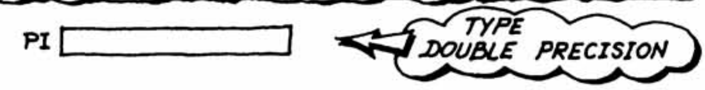


The size and precision of numbers that may be kept in these boxes varies from computer to computer. The magnitude of a typical *REAL* may lie between -10^{38} and $+10^{38}$ and be represented to a precision of at least six significant decimal digits. A typical *INTEGER* may take any value from −32767 to +32767. On some computers these sizes and precisions are much greater.

Six significant decimal digits may not offer enough precision in some calculations. For such cases Fortran provides *DOUBLE PRECISION* variables which may have the same size range as ordinary *REAL* variables (typically $\pm 10^{38}$) but a precision of at least eleven significant decimal digits.

```
DØUBLE PRECISIØN  PI
PI = 3.14159265358979 3D0
```

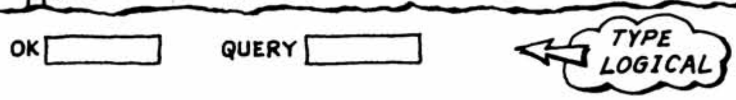
SEE PAGE 24



The simplest type in Fortran is type *LOGICAL*. A logical variable may take only two values: *true* or *false*.

```
LØGICAL  ØK, QUERY
ØK = .TRUE.
QUERY = .FALSE.
```

SEE PAGE 24



The most complicated type in Fortran is type *COMPLEX*. Every complex variable consists of two parts: the *real* and the *imaginary*. Each part has the same range and precision as a *REAL* variable on the same computer.

```
CØMPLEX  PØTEN
PØTEN = ( 123.45 , -6.0 )
```

123.45 − 6i SEE PAGE 25

Fortran 66 refers to *storage units* of which:

- *one* is needed to store an *INTEGER*, a *REAL*, a *LOGICAL* variable
- *two* are needed to store a *DOUBLE PRECISION* or *COMPLEX* variable

but in practice different Fortrans employ computer "words" in entirely different proportions. Do not rely, for example, on a *REAL* variable and an *INTEGER* variable occupying the same space: *they often don't*.

All variables may have their types *declared* as illustrated below:

```
      REAL   X, Y, Z, AA
      INTEGER  I, IA, KØUNT
      DØUBLE  PRECISIØN    PI, DX4
      LØGICAL   ØK
      CØMPLEX   PØTEN
```

but it is not essential to declare *INTEGERS* or *REALS*. When the type of a variable is not explicitly declared it is *implicitly* declared as type *INTEGER* if its initial letter is:

I , J , K , L , M , N

otherwise it is implicitly declared as type *REAL*. This convention is sometimes called *implicit typing*.

The forms of type statement are:

REAL *name, name, ..., name*
INTEGER *name, name, ..., name*
DOUBLE PRECISION *name, name, ..., name*
LOGICAL *name, name, ..., name*
COMPLEX *name, name, ..., name*

where:

name is a symbolic name of a variable or of an array (see Chapter 5). When a symbolic name refers to an array then the *dimensions* of that array may also be given as though the name were in a list following a *DIMENSION* statement

When a variable is named as above this does not imply that it has an initial value of zero; its content is completely *undefined*; there is no saying what the little box might originally contain.

Some Fortrans (including Fortran 77) provide a statement called *IMPLICIT* using which the programmer may alter the range of initials denoting types of variable. This facility should not be used in a program intended to be portable.

Some Fortrans (including Fortran 77) provide two kinds of integer : *short* and *long* . A short integer is denoted *INTEGER*2* because it consists of two "bytes" each of eight binary digits; its range is $\pm 2^{15}-1$. Similarly the long integer is denoted *INTEGER*4* and its range is $\pm 2^{31}-1$. A program that relied upon the long integer would not be fully portable.

Fortran 77 provides an extra type called *type CHARACTER*. If a program is to be portable, however, it is better to manipulate characters by storing them in variables of type *INTEGER* as discussed in Chapter 9.

Thus there are just *five* types of variable in Fortran 66. There are also the same five types of array (Chapter 5) each being an array of *elements*. In general what can be done with a variable can also be done with an *array element* of like type. There are, however, some important exceptions to this rule and attention is drawn to such exceptions wherever they occur.

TYPES OF CONSTANT

REAL, INTEGER, DOUBLE PRECISION, LOGICAL, HOLLERITH, COMPLEX

The introductory example had the line:

```
TØP = 3.14 * (DIAM **2) / 4.0
```

REAL CONSTANT   REAL CONSTANT

where the arrowed items are *constants* of type *REAL*. *REAL* constants are written with decimal points. Thus a *REAL* constant with a value of two may be written 2.0 or even as 2. but never as 2 because 2 is an *INTEGER* constant as explained below. A *REAL* constant of a half may be written as 0.5 or just .5 without the leading zero.

A *REAL* constant may also be written in *exponent form* as illustrated below:

```
TØP = 0.314E1 * ( DIAM**2) / 400.0E-2
```

REAL CONSTANT   REAL CONSTANT

where letter *E* says "times ten to the power of..." then must come an integer to specify the power. The example above shows *0.314E1* which means 0.314×10^1 and *400.0E-2* which means 400.0×10^{-2}. One million could be written in many ways including *1.0E6*, *10.0E+5*, *10.E5*, *10E5*. Notice there need not be a decimal point in a *REAL* constant if written in exponent form (*10E5*).

The introductory example had the lines:

```
NPØTS = INT(PØTS) + 1
WRITE (6, 200)  NPØTS
```

INTEGER CONSTANT

INTEGER CONSTANTS

where the arrowed items are *constants* of type *INTEGER*. *INTEGER* constants consist only of digits. There should be no decimal point and no letter *E*. An *INTEGER* constant of thirty thousand is written *30000* and not *3E4* which is one of the forms of a *REAL* constant.

A *DOUBLE PRECISION constant* has the same form as a *REAL* constant *written in exponent form* except that letter *D* replaces letter *E*.

```
DØUBLE PRECISIØN  DIAM, TØP
TØP = 3.14159265358979 3D0 * (DIAM**2)/4.0D0
```

DOUBLE PRECISION CONSTANT   DOUBLE PRECISION CONSTANT

So far constants of type *INTEGER*, *REAL* and *DOUBLE PRECISION* have been defined without mention of a sign (+ or −) as a prefix. Without the sign these constants are said to be *unsigned* : with a preceding + or − they are said to be *signed*. If there is no mention of *signed* or *unsigned* then the sign is optional. For example "An integer constant of two" may be written 2 or +2 ; "An integer constant of minus two" should, of course, be written −2 .

There are only two constants of type *LOGICAL*. These are written *.TRUE.* and *.FALSE.* as illustrated below:

```
LØGICAL    ØK, QUERY
ØK = .TRUE.
QUERY. =  .FALSE.
```

LOGICAL CONSTANTS

Hermann Hollerith pioneered punched-card machines in the 1890s. His initial, *H*, survives in Fortran 66 to denote a sequence of characters as illustrated in the introductory example:

```
200   FØRMAT (1X, 9H YØU NEED, I2, 5H PØTS )
```

HOLLERITH LITERAL HOLLERITH LITERAL

The unsigned integer before letter *H* specifies the number of characters in the *literal* immediately following the *H*. This number includes spaces. In the above example there is a space before and after *YOU* and a space before *POTS*. In a *FORMAT* statement these items are called *Hollerith literals*. There are also, in Fortran, *Hollerith constants* which have precisely the same form as Hollerith literals:

```
2HAB
```

"AB"

but may be stored in variables of type *INTEGER*. In fact Fortran 66 permits Hollerith constants to be stored in variables of any type and specifies no limit to their length ⇔ length being a property set by the particular computer being used. To make a program fully portable, however, confine Hollerith constants to *INTEGER* variables and do not store more than two characters per variable. Storage of characters is introduced in Chapter 9, and an example of their manipulation in Chapter 12. The examples should demonstrate that the restrictions advocated above do not prevent the programmer from manipulating characters effectively.

We anticipate Chapter 9 with the following example by which the Hollerith constant 2*H*AB (*i.e.* the letters AB) may be stored in the integer variable named K:

```
      INTEGER  K
      DATA     K / 2HAB /
```

HOLLERITH CONSTANT AB

and this may **NOT** be achieved by assignment:

```
      K = 2HAB
```

Some Fortrans allow Hollerith literals and constants to be written between quotation marks ⇔ dispensing with the count and letter *H*:

```
200   FØRMAT (1X, ' YØU NEED', I2, ' PØTS' )
```

which is certainly more elegant (and is the form preferred by Fortran 77) but should not be used if a program is to be fully portable. There is no quotation mark in the Fortran 66 character set.

Constants of type *COMPLEX* are written as two *REAL* constants separated by a comma and enclosed between parentheses:

```
      CØMPLEX  C
      C = ( 2.7 , -0.9 )
```

COMPLEX CONSTANT

where the 2.7 constitutes the real part and the −0.9 constitutes the imaginary part. Mathematicians would write the above complex constant as $2.7-0.9i$ where the i stands for $\sqrt{-1}$ (the square root of minus one). Non-mathematicians need not bother with type *COMPLEX*.

ARITHMETIC EXPRESSIONS

REAL, INTEGER, DOUBLE PRECISION, COMPLEX

The introductory example showed:

```
      TØP = 3.14 * (DIAM**2) / 4.0
      WALL = 3.14 * DIAM * HEIGHT
      PØTS = (TØP + WALL) / CØVRG
```

where in each case the *expression* on the right of the equals sign is evaluated and the result is assigned to the variable nominated on the left.

Expressions may consist of variables, constants and other terms* bound together with *operators*. The *arithmetic* operators in Fortran are:

- **+** for addition – or as a prefix to denote a positive quantity (*e.g. A=+B*)
- **−** for subtraction – or as a prefix to denote a negated quantity (*e.g. I=-J*)
- ***** for multiplication
- **/** for division
- ****** for exponentiation

An expression is evaluated generally from left to right in three sweeps. Exponentiation is done in the first sweep; multiplications and divisions (with equal precedence) in the second sweep; additions and subtractions (with equal precedence) in the third sweep.

Parentheses may be used to emphasize or change the natural precedence described above – and achieve what one expects from the rules of algebra. Notice that the parentheses in the first expression above are not necessary: *DIAM**2* would automatically be taken first. But parentheses are vital in the third line: *PØTS = (TØP + WALL)/ CØVRG* .

In general, all terms in an expression should be of the same type. There is a notable exception in the case of exponentiation illustrated in the introductory example:

```
                           INTEGER
      TØP = 3.14 * (DIAM**2) / 4.0
                    REAL
```

where *DIAM* is of type *REAL* and the power 2 is of type *INTEGER*. Fortran 66 allows an arithmetical term of any type (*INTEGER, REAL, DOUBLE PRECISION* or *COMPLEX*) to have an *INTEGER* exponent; the result of such exponentiation having the same type as the term being exponentiated.

In the above example it would be correct to have *DIAM**2.0* in place of *DIAM**2* but this might force the Fortran compiler to evaluate this term by generating logarithms instead of multiplying *DIAM* by *DIAM*.

*the other terms are "function references" (illustrated opposite) and "array elements" (Ch. 5).

A *DOUBLE PRECISION* term may be exponentiated using a *REAL* exponent or a *DOUBLE PRECISION* exponent ≈ the result being *DOUBLE PRECISION* in each case:

```
INTEGER I                  INTEGERS ALWAYS ALLOWABLE
REAL    R                  (SEE RULE OPPOSITE)
DØUBLE PRECISIØN   D, DBL, DA
DA = DBL**I + DBL**R + DBL**D
```

Fortran 66 also allows combination of *REAL* and *DOUBLE PRECISION* terms in other ways ≈ the result being *DOUBLE PRECISION*. Fortran 66 also allows combination of *REAL* and *COMPLEX* terms ≈ the result being *COMPLEX*. Examples follow.

```
REAL     R
DØUBLE  PRECISIØN   DBL, D
CØMPLEX             CMPLX, C
DBL  =  R*D + D/R
CMPLX  =  R*C + C/R
```

Although many Fortran compilers permit other mixtures of type their details and interpretations differ. So for the sake of portability do not take advantage of "mixed mode" facilities apart from those just described.

The introductory example showed:

```
               REAL
NPØTS  =  INT (PØTS)  +  1
          INTEGER        INTEGER
```

where *INT( )* is a Fortran *function* for converting a *REAL* quantity into an *INTEGER* by truncation (for example *INT(3.995)* yields an integral value of 3). Thus both terms in the expression above are of type *INTEGER*. Fortran provides a range of such functions for converting expressions and parts of expressions from one type to another. These functions (defined in Chapter 6) make it unnecessary to rely on non-standard "mixed mode" facilities.

There are some pitfalls to be avoided when writing expressions. In dividing integers any fractional part is lost. In the example below *I* and *J* would be assigned values of 2 and −2 respectively:

```
I =  7/3
J =  -7/3
```

and there is no implied multiplication. The expression below should have an asterisk between the two terms:

```
A = (C-D)(E-F)
```

and it is not permitted to have two operators together without a bracket intervening. The expression below should read *B*(-C)* or *-B*C* :

```
A = B*-C
```

Some expressions might be ambiguous to the reader:

```
A  = B/C/D
P  = Q**R**S
X  = -Y**Z
```

There is a world of difference between *8/(4/2)=4* and *(8/4)/2=1*. Fortran 66 should treat the first example as *A=((B/C)/D)* but there might be a compiler somewhere that would treat it the other way. The second example illustrates a forbidden form, but either *(Q**R)**S* or *Q**(R**S)* would be allowed. Fortran should treat the third example as *X=(-(Y**Z))* but there might be a compiler that would try evaluating X as *(-Y)**Z* and get into trouble. ***When in doubt use brackets!***

LOGICAL EXPRESSIONS

TYPE LOGICAL: TRUE OR FALSE
(SKIP THIS PAGE ON FIRST READING)

An expression of type *LOGICAL* is one that has the Boolean value *true* or *false*. The main use of logical expressions in Fortran is in the *Logical IF* statement described in Chapter 4 – hence the invitation to skip this double page on first reading.

The simplest logical expressions are the logical constants:

```
.TRUE.
.FALSE.
```

but a logical expression may be more complicated – consisting of *relational expressions* bound together using *logical operators*.

A relational expression consists of two arithmetic expressions bound together by one of the following six *relational operators*:

.LT.	meaning	"less than"
.LE.	meaning	"less than or equal to"
.EQ.	meaning	"equal to"
.NE.	meaning	"not equal to"
.GT.	meaning	"greater than"
.GE.	meaning	"greater than or equal to"

An example of a relational expression involving type *REAL* is:

```
A*B/C  .LE.  X+5.0
```

which takes the Boolean value *true* if the numerical value of *A*B/C* turns out to be less than or equal to the numerical value of *X+5.0*. If the value of *A*B/C* turns out *not* to be less than or equal to that of *X+5.0* the expression takes the Boolean value *false*.

An example of a relational expression involving type *INTEGER* is:

```
I*2  .EQ.  J
```

There is no such thing as a relational expression involving type *COMPLEX*.

It is wrong to compare an expression of type *INTEGER* with an expression of any other type:

```
I*2 .GT. X+5.0
```

however it is admissible in Fortran 66 to compare an expression of type *REAL* with an expression of type *DOUBLE PRECISION*:

```
X  .LT.  1.5D6
```

but such practice can be dangerous because you may not be able to tell in which mode the comparison is made. Is X converted to double precision before comparing with 1.5D6 ? Or is 1.5D6 converted to standard precision before being compared with X ? In such cases it is best to use one of the intrinsic functions (Chapter 6) to make certain about the mode of comparison:

```
DBL(X)  .LT.  1.5D6
```

CONVERTS X TO DOUBLE PRECISION

The simplest logical expressions (as said before) are the logical constants *.TRUE.* and *.FALSE.* — but the expressions may be much more complicated, consisting of relational expressions (defined opposite) bound together with *logical operators*. The three logical operators are:

.OR.	for	" logical disjunction "
.AND.	for	" logical conjunction "
.NOT.	for	" logical negation "

Examples of logical expressions are:

```
.TRUE.
.NØT. .TRUE.
I*2   .EQ.  J
I .EQ. J   .ØR.   I .EQ. K
```

A RELATIONAL EXPRESSION ON ITS OWN

RELATIONAL EXPRESSIONS BOUND TOGETHER WITH A LOGICAL OPERATOR

In the fourth example immediately above, the logical expression takes the Boolean value *true* if *I* equals *J* or if *I* equals *K* or if *I, J, K* are all equal to one another. In other words the *.OR.* is *inclusive* rather than *exclusive*.

The binding strength of *.NOT.* is stronger than that of *.AND.* which, in turn, is stronger than that of *.OR.* ; thus:

```
.NØT.  BLACK  .AND.  WHITE
```

means *(.NOT. BLACK) .AND. WHITE* rather than *.NOT.(BLACK .AND. WHITE)* .

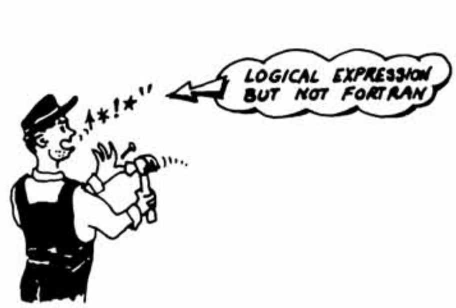
Brackets may be used in the construction of logical expressions (and to add clarity to expressions that do not really need brackets) just as with arithmetic expressions:

```
( X .GE. 1.0) .AND. ( X .LE. 2.0 )
```

This expression takes the value *true* if the value stored in variable X lies between 1.0 and 2.0 .

In the absence of brackets Fortran deals with relational expressions first, then applies the logical operators (first *.NOT.* then *.AND.* then *.OR.*) to the resulting Boolean values. Make sure you don't leave a relational expression "dangling" :

```
X  .GE.  1.0  .AND.   .LE.  2.0
```



ASSIGNMENT

STATEMENTS WITH AN EQUALS SIGN
(EVERY ASSIGNMENT IS AN EXECUTABLE *STATEMENT)*

The introductory example showed several *assignments* :

```
      TØP = 3.14 *(DIAM**2)/4.0        REAL
      WALL = 3.14 * DIAM * HEIGHT      ASSIGNMENTS
      PØTS = (TØP + WALL)/ CØVRG
      NPØTS = INT(PØTS) + 1            INTEGER ASSIGNMENT
```

where the arithmetic expression on the right of the equals sign is evaluated and the result assigned to the variable nominated on the left ~ obliterating any previous value found there.

The first three assignments above are of type *REAL* : the result of a *REAL* expression being assigned to a *REAL* variable. The last assignment above is of type *INTEGER* : the result of an *INTEGER* expression being assigned to an *INTEGER* variable. The piece of program below illustrates *DOUBLE PRECISION*, *COMPLEX* and *LOGICAL* assignments:

```
      DØUBLE PRECISIØN   D, DBL
      CØMPLEX      C, CMPLX
      LØGICAL    ØK                 DOUBLE PRECISION
      DBL = 1D6 * D                 ASSIGNMENT
      CMPLX = (1.5, -1.0) * C       COMPLEX ASSIGNMENT
      ØK = .TRUE.                   LOGICAL ASSIGNMENT
```

For the sake of clarity it is best to ensure the expression on the right of the equals sign is of the same type as the variable* on the left of it. However, Fortran 66 does permit a change of type across the equals sign in the case of *INTEGER*, *REAL* and *DOUBLE PRECISION*. The allowable forms of assignment are:

*INTEGER variable** *REAL variable** *DOUBLE PRECISION variable**	=	*INTEGER expression* *REAL expression* *DOUBLE PRECISION expression*
*COMPLEX variable**	=	*COMPLEX expression*
*LOGICAL variable**	=	*LOGICAL expression*

In the first of the three forms depicted above the Fortran compiler automatically converts the result of the expression (if necessary) to the type of variable to which the value is assigned. Thus it *would* be permissible to write:

```
      NPØTS = PØTS + 1.0
      INTEGER REAL  REAL
```

in place of:

```
      NPØTS = INT(PØTS) + 1
      INTEGER INTEGER    INTEGER
```

Warning! Do not assume the computer would preserve the full accuracy of the product in an assignment such as:

```
      D = RA * RB
```

where *D* is a *DOUBLE PRECISION* variable and *RA* and *RB* are *REALS*. But you may ensure full accuracy by changing the form of the assignment to:

```
      D = DBL(RA) * DBL(RB)      SEE CHAPTER 6 FOR DBL()
```

* *"variable" is here taken to include "array element".*

LOANS

AN EXAMPLE TO ILLUSTRATE ARITHMETIC ASSIGNMENTS

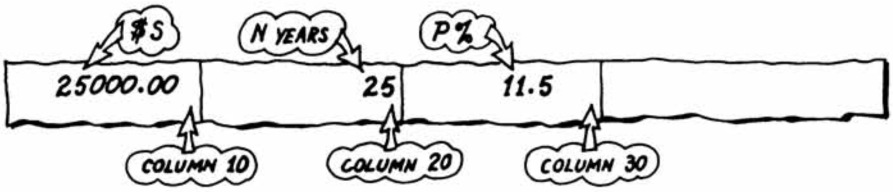
The monthly repayment, \$C, on a mortgage loan of \$S at P% compound interest over N years is given by:

$$C = \frac{SR(1+R)^N}{12\left[(1+R)^N - 1\right]}$$

where:

$$R = P \div 100$$

Here is a set of data for a program designed to compute C:



And here is the program itself:

```
      READ(5, 100)   S, N, P
      R =  P/ 100.0                        <- ASSIGNMENTS
      C = S*R*(1.0+R)**N /(12.0*((1.0+R)**N - 1.0))
      WRITE (6, 200)   C
      STØP
100   FØRMAT (F10.0 ,  I10 ,  F10.0)
200   FØRMAT(1X, 19H  MØNTHLY  BURDEN = $, F6.2)
      END
```

And here is the resulting output:

```
MONTHLY BURDEN = $256.45
```

Some innocent looking assignments can cause unexpected trouble:

```
      A = B/C
      X = Y**Z
```

If variable C holds zero — or a number near to zero — there would probably be an error message printed and execution would cease. Variable X would be set to 1.0 if Z were zero and Y greater than zero (*e.g.* X = 6.5**0) but if Y contained a value of zero or less the program would probably fail. A negative value cannot be raised to a non-integral power — this is a mathematical rule having nothing to do with Fortran specifically.

$$(-6.5)^{2.1}$$

EXERCISES

CHAPTER 3

3.1 Read what your manual says about the set of allowable characters and strike out those not in the standard set of 47.

3.2 If your manual permits names to consist of more than six characters, make a marginal note that no more than six should ever be used.

3.3 Note in the table below the size and range of each type of variable on the computer to which you have access

type	number of bits	range	precision
INTEGER			
REAL			
DOUBLE PRECISION			
LOGICAL			
COMPLEX			

3.4 "Fortran" is an acronym for "*for*mula *tran*slation". Take a formula used in your own field of expertise. Write a small program (like the one on the previous page) to read data; evaluate your chosen formula; print the result.

4

CONTROL WITHIN A PROGRAM UNIT

SIMPLE LOOPS
SHAPES (OR STRUCTURES)
LOGICAL IF
UNCONDITIONAL TRANSFER
COMPUTED GO TO
CONTINUE
THE DO LOOP
ARITHMETIC IF
ASSIGNED GO TO
AREAS OF SHAPES (AN EXAMPLE)
EXERCISES

SIMPLE LOOPS

INTRODUCING THE GO TO *AND* LOGICAL IF *STATEMENTS*

Here is an example of a program that does nothing but read an integer, print it, then repeat the process until there are no more integers to read – when an error message from the computer would be printed to indicate an unsatisfied *READ* statement.

```
 10  READ (5, 100)   INT
     WRITE (6, 200)  INT
     GØ  TØ   10
     STØP
100  FØRMAT  (I10)
200  FØRMAT  (1X, I10)
     END
```

AN "INFINITE" LOOP

Notice the *GO TO* statement which causes control to pass to the statement having the nominated label. This particular *GO TO* would prevent the *STOP* statement ever being obeyed.

If the first item of data were a *count* of the number of integers following:

```
   3   ← COLUMN 10
 125
2793
  43
```

SAYS THAT 3 INTEGERS FOLLOW IN THIS CASE

then the program could be rewritten as follows:

```
     READ (5, 100)   KØUNT
 10  IF   (KØUNT .EQ. 0)  GØ TØ  20
     READ (5, 100)   INT
     WRITE  (6, 200) INT
     KØUNT = KØUNT - 1
     GØ  TØ   10
 20  STØP
```

A "WHILE" LOOP

and this program would stop properly even if the count of integers were zero.

Alternatively the program could be written like this:

```
     READ  (5, 100)  KØUNT
 10  READ  (5, 100)    INT
     WRITE    (6,200)  INT
     KØUNT  =  KØUNT  - 1
     IF  ( KØUNT   .NE.  0)    GØ  TØ  10
     STØP
```

A "REPEAT UNTIL" LOOP

but this program would fail if the count of the number of following integers were zero. Statement 10 cannot be avoided so there must be at least one integer to be read and printed.

Another way to tackle the problem is to omit the count of integers from the data but terminate the list of integers with a zero (assuming, in this case, that the program is designed to handle only non-zero integers). The program could be written like this:

```
 125
2793
  43
   0  ← TERMINATOR
```

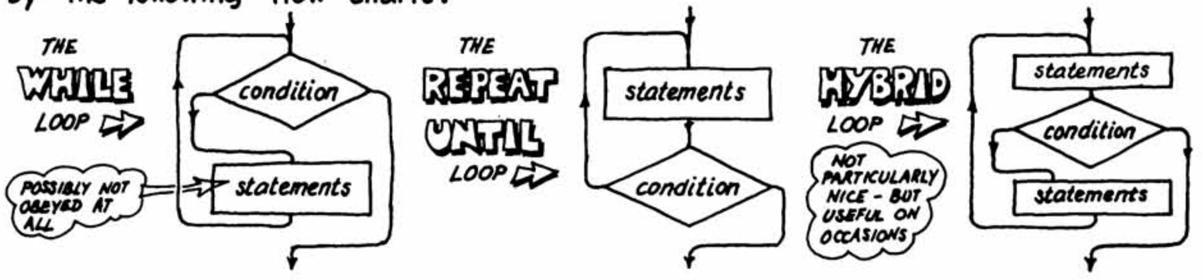
```
 10  READ (5, 100)  INT
     IF  ( INT  .EQ.  0 )   GØ  TØ  20
     WRITE  (6,  200)   INT
     GØ  TØ   10
 20  STØP
```

A "HYBRID" LOOP

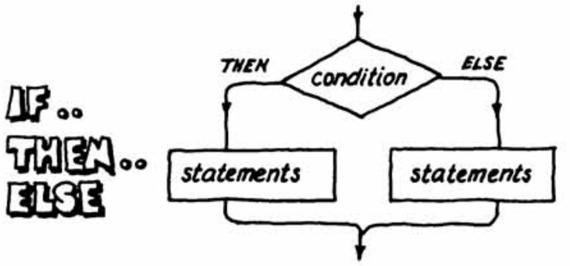
SHAPES (OR "STRUCTURES")

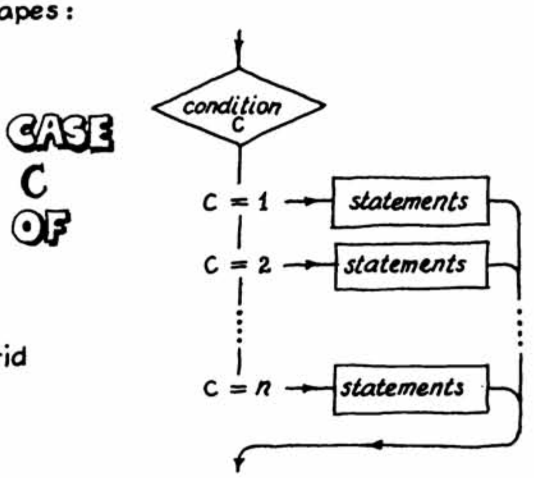
STICK TO THESE SHAPES OTHERWISE YOU RISK GETTING INTO A MESS

The last three programs opposite display the shapes (or *structures*) defined by the following flow charts:



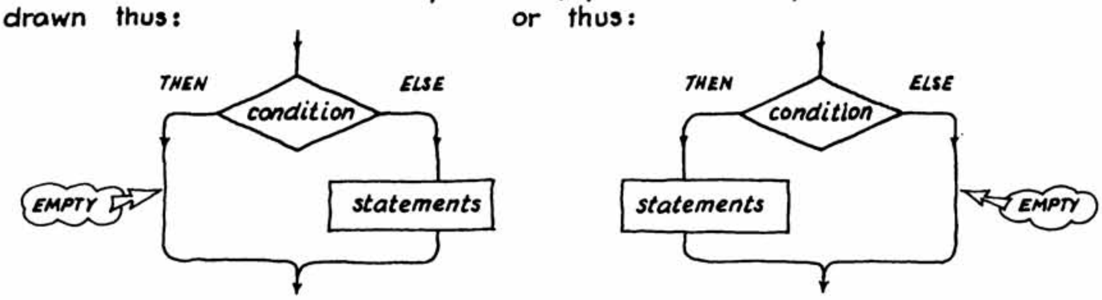
Add to these loops the following two shapes:



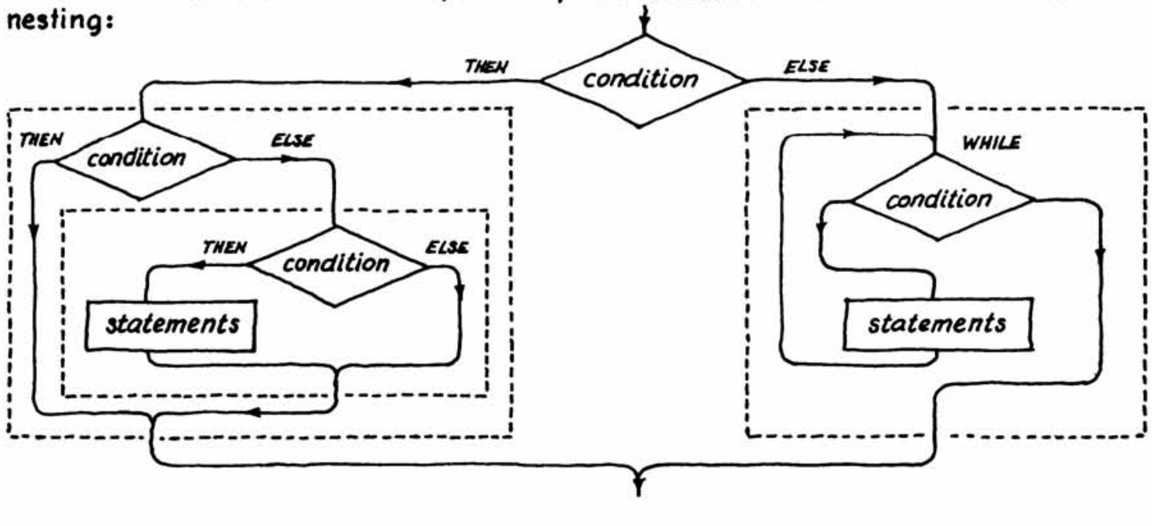


Using these shapes (even omitting the hybrid loop) it is possible to program anything capable of being programmed. And the program would be properly *structured*.

The box called *statements* may be empty. For example *IF..THEN..ELSE* may be drawn thus: or thus:



Conversely any box called *statements* may contain very many Fortran statements to be obeyed – *including those comprising the shapes illustrated*. In other words all these shapes may be *nested*. Here is an illustration of nesting:



LOGICAL IF *THE MOST USEFUL CONTROL STATEMENT*

Fortran offers some special methods of control (considered later) but the shapes previously sketched may all be achieved using the LOGICAL IF statement. Its form is:

IF (*logical expression* **)** *statement*

where:

logical expression	is an expression whose value is either *true* or *false* as described on pages 28. and 29.
statement	need not be the GO TO statement previously illustrated. It may be any executable statement other than another LOGICAL IF or a DO statement (see page 40).

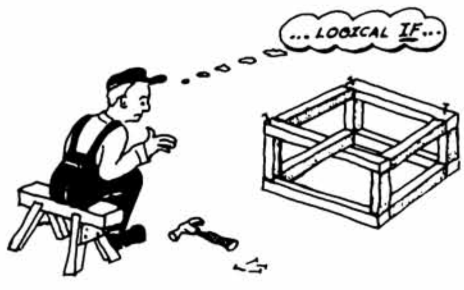
Here are some examples of LOGICAL IF statements:

```
      IF (ØK) GØ TØ 66
      IF (I .NE. J) STØP
      IF (A*B/C .LE. X+5.0) A = A + 1.0
   66 IF ((X .GT. Y) .ØR. (X .LT. SQRT(Z))) I=J
```

When such a statement is obeyed the logical expression is first evaluated. If the result is the Boolean value *true* then the *statement* incorporated in the LOGICAL IF statement is obeyed. If the Boolean value turns out to be *false* then control passes straight to the statement immediately following the LOGICAL IF.

A subtle point arises when the *logical expression* is evaluated. In the statement labelled 66 above the computer might compare values of X and Y; discover that X was greater than Y; and so obey the statement *I=J* without bothering to compute the square root of Z. That's all right. But if the function were one devised by the programmer (Chapter 7) and which *changed the value held in J*, what then? The result would probably vary from one Fortran to another. Never rely on such an expression being fully evaluated.

Opposite is a program designed to illustrate some nested "shapes" previously defined. The program reads a number of integers to discover the maximum and minimum. The data are written after a count of the number of integers to follow – precisely as in an earlier example.



Here are the data:

```
         4
       217
       -73
         0
        46
```

COUNT OF INTEGERS FOLLOWING (AT LEAST ONE)

COLUMN 10

Here is the program. It will cope with any number of integers but no integer may exceed a million in absolute value:

```
      INTEGER     MAX, MIN, INT, KØUNT
      READ  (5, 100)  KØUNT
      MAX  =  -1.0E6
      MIN  =  +1.0E6
   10 READ  (5, 100)  INT
      IF  (INT  .GT.  MAX)   MAX = INT
      IF  (INT  .LE.  MIN)   MIN = INT
      KØUNT  =  KØUNT  -  1
      IF  (KØUNT  .NE.  0)   GØ  TØ  10
      WRITE  (6, 200)  MAX,  MIN
      STØP
  100 FØRMAT  (I 10)
  200 FØRMAT  (1X, 4H MAX, I7, 4H MIN, I7)
      END
```

REPEAT UNTIL

NO ITEM OF DATA MAY FALL OUTSIDE THE RANGE ± 1,000,000 IN THIS PROGRAM

IF..THEN..ELSE NESTED

INTEGERS IN A FIELD OF 10 COLUMNS

BLANK

BLANK

And the output would be:

```
 MAX    217 MIN    -73
```

UNCONDITIONAL TRANSFER *GO TO , STOP, PAUSE*

Thus far this chapter has introduced the *GO TO* statement by example only. The form of this statement is:

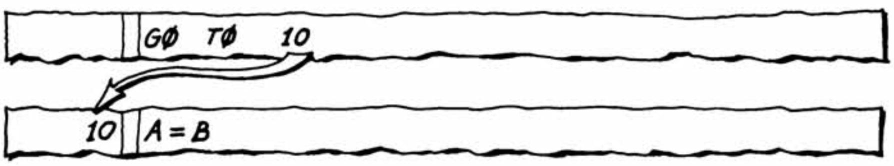
GO TO *label*

OPTIONAL SPACE

where:

label is the label of an executable statement in the same program unit as the *GO TO*. This item should consist only of digits (the name of an integer variable is not allowed).

When this statement is obeyed control is passed to the statement bearing the nominated label:



The introductory example illustrated the *STOP* statement. Its form is:

STOP *octal number*

where:

octal number is optional. If included it should be limited in length to 5 octal digits. (Octal digits are those from 0 to 7, but Fortran 77 permits decimal digits from 0 to 9)

When this statement is obeyed execution of the program ends normally. It depends on the computer installation what happens to the octal number, but that number should somehow be conveyed to the person trying to run the program — perhaps by being sent to unit 6 which is conventionally the line printer. There may be any number of *STOP* statements in a program; the programmer might be interested to know which of them caused his program to stop.

Notice that the *STOP* statement is the standard way to cause a program to stop execution. The *END line* has an entirely different purpose as explained on page 14. The *CALL EXIT* statement — available in many Fortrans — is *not* a statement defined by Fortran 66 or Fortran 77.

Fortran 66 and Fortran 77 both recognize a *PAUSE* statement: the word *PAUSE* optionally followed by an integer (*octal* in Fortran 66). This statement may have been useful in the days when the programmer had the computer all to himself. He could note the octal number displayed by the computer and decide whether or not to cause the program to start execution again — carrying on from the statement immediately following the *PAUSE*. But on large modern computers the *PAUSE* statement is either ignored or dealt with according to rules local to the installation. So the *PAUSE* statement is best avoided if programs are to be made fully portable.

COMPUTED GO TO *"CASE C OF ..."*

Fortran provides a statement called the *computed GO TO* which causes control to pass to one of several labelled statements according to the value currently stored in an integer variable. The form is:

GO TO ***(label, label, label, ... ,label), variable***

FIRST SECOND THIRD LAST N.B.

where:

label is the label of some executable statement in the same program unit as the *computed GO TO*. There should be one or more of these inside the parentheses. Each may consist only of digits; the name of an integer variable is not allowed.

variable is the name of a variable of type *INTEGER*. (An array element is not allowed here.)

When this statement is obeyed the value stored in the integer variable is consulted. If this value is *unity*, control passes to the statement labelled with the *first* label written between the parentheses; if the value is *two*, control passes to the statement labelled with the *second* label between the parentheses ~ and so on.

```
      GØ TØ (10, 10, 20, 10) ,   JMP
   10 A = X
      GØ TØ  30

   20 A = Y
   30 CØNTINUE
```

In this example control would pass to statement *10* if the value stored in *JMP* were 1 or 2 or 4 , but to statement *20* if the value in *JMP* were 3 .

What if the value in *JMP* turned out to be negative or greater than 4 ? The answer is that different Fortrans do different things. It is best to *protect* the *computed GO TO* as illustrated below.

```
      IF ((JMP .GT. 4) .ØR. (JMP .LT. 1)) STØP 077
      GØ TØ (10, 10, 20, 10) ,   JMP
```

077: SEE OPPOSITE PAGE

CONTINUE *AN EXECUTABLE STATEMENT ALWAYS LABELLED*

The *CONTINUE* statement is a "do nothing" statement of the form:

CONTINUE

This statement is classed as an *executable* statement even though it causes no work to be done. It is a useful statement ~ when labelled ~ to act as the terminal statement of a *DO loop* (page 40) or as a point in a program where several paths converge, as in the example above at label 30 .

Although Fortran 66 does not forbid it, a *CONTINUE* statement without a label is pointless ~ and probably points to a mistake in the program.

THE DO LOOP "REPEAT UNTIL"

Fortran provides a special form of *repeat until* structure called the *DO loop*. Its form is:

DO *label control = initial, terminal, increment*

or:

DO *label control = initial, terminal*

where:

label is the label of some executable statement physically following the *DO*. This item must consist only of digits.

control is the name of an integer variable (not an array element).

initial is either an unsigned integer or the name of an integer variable (not an array element) which must hold a value greater than zero when the *DO* is executed.

terminal is of the same form as *initial* and must hold a value greater than or equal to that of *initial* when the *DO* is executed.

increment is of the same form as *initial* and *terminal* and must have a value greater than zero when the *DO* is executed. If omitted, *increment* is assumed to be unity.

Here is part of the "max. & min." program again, but using the *DO loop*:

```
      READ (5,100) KØUNT
      MAX = -1.0E6
      MIN = +1.0E6
      DØ 10 I = 1, KØUNT
      READ (5,100) INT
      IF (INT .GT. MAX) MAX = INT
   10 IF (INT .LE. MIN) MIN = INT
      WRITE (6, 200) MAX, MIN
```

On meeting the *DO* the computer assigns the *initial* value to *control* (in this case the integer variable *I* is assigned a value of *1*). Then the computer obeys subsequent instructions down to and including the one with the nominated label (in this case the statement labelled *10*).

Then the *control* variable is incremented by the value held in *increment* (in this case unity by default). If the augmented control variable now holds a value greater than that specified by *terminal* the loop is satisfied and control passes to the statement immediately following the labelled one; otherwise the loop is executed again. The word *DO* is short for "ditto".

Note carefully the restrictions on the components of a *DO*. It is wrong to count backwards:

```
      DØ 10 I = KØUNT, 1, -1
```

and it is wrong to count from zero:

```
      DØ 10 I = 0, 9
```

although there are some Fortrans that permit such abuses. Many Fortrans obey any *DO loop* they encounter at least once — even when the controlling parameters are in conflict:

```
      DØ 10 I = 10, 9
```

The *label* must be the label of an executable statement other than:
(i) *arithmetic IF* (page 42), (ii) *RETURN* (page 66), (iii) *STOP*, (iv) *PAUSE*, (v) *GO TO*
or (vi) another *DO* statement. The terminating statement may, however, be a *logical IF* (as already illustrated) provided that this does not incorporate any of the forbidden statements (i) to (vi) set out above.

```
      DØ 10 I = K, L, M
   10 IF (A .EQ. B) GØ TØ 20
```

INCORPORATES A FORBIDDEN STATEMENT

Many programmers make it a rule to finish every *DO loop* at a *CONTINUE*:

```
      DØ 10 I = 1, KØUNT
      READ (5, 100) INT
      IF (INT .GT. MAX) MAX = INT
      IF (INT .LT. MIN) MIN = INT
   10 CØNTINUE
```

Never tamper with controlling parameters inside a loop:

```
      DØ 20 J = K, L, M
      M = 2*M
      J = J + 1
```

It is safe to *use* the value of the control variable inside a loop:

```
      DØ 10 K = 1, 12
      L = 5 * K
   10 WRITE (6, 300) K, L
  300 FØRMAT (1X, I3, 10H TIMES 5 =, I3)
```

PRINTS THE 5 TIMES TABLE

And it is safe to jump out of a loop before it has run its course ≈ in which case the control variable retains its current value:

```
      DØ 10 I = 1, 100
      READ (5, 400) X
      IF (X .EQ. 0.0) GØ TØ 20
      WRITE (6, 500) X
   10 CØNTINUE
      STØP
   20 WRITE (6,600) I
```

VARIABLE I WILL CONTAIN THE NUMBER OF NUMBERS READ – HAVING FOUND A ZERO AMONG THE DATA

But never assume anything about the value of a control variable once the loop has run its course. In the example below the printed value of *K* might turn out to be 4 ; perhaps 3 ; perhaps some arbitrary value.

```
      DØ 10 K = 1, 3
   10 CØNTINUE
      WRITE (6, 600) K
```

?

Some Fortrans allow a jump *back* into the middle of a loop ≈ the statements executed whilst being out of the loop constituting an "extended range". But some Fortrans (including Fortran 77) object to this practice so it is best not to employ extended ranges. Never jump into the middle of a *DO* loop!

```
      DØ 10 I = 1, 3
      GØ TØ 20
   10 CØNTINUE

   20 J = I + 1
      GØ TØ 10
```

EXTENDED RANGE ?

ARITHMETIC IF

NOW LARGELY SUPERSEDED BY THE LOGICAL IF *STATEMENT*

Fortran provides a unique *three-way switch* of the form:

IF (*expression* **)** *label* **,** *label* **,** *label*

where:

expression is an arithmetic expression of type *INTEGER* or *REAL* or *DOUBLE PRECISION*

label is the label of an executable statement in the same program unit as the *IF* statement

When this statement is obeyed the expression is evaluated: the result will be negative or precisely "zero" or positive. Control then passes to the statement having the first, second or third label respectively according to this result.

```
      IF  (X + Y)     10, 10, 20
```

```
   10 I = J              OBEYED NEXT IF
      GØ  TØ   30        (X+Y) ≤ 0.0
```

```
   20 I = K              OBEYED NEXT IF
   30 CØNTINUE           (X+Y) > 0.0
```

The "zero" referred to above is the kind of zero appropriate to the expression being evaluated. Thus the zero would be:

0.0	for a *REAL* expression (*e.g.* (X+Y))
0	for an *INTEGER* expression (*e.g.* (M+N))
0.0D0	for a *DOUBLE PRECISION* expression (*e.g.* (DX+1.0D0))

The *arithmetic IF* was once the only *IF* statement in Fortran. Its use has been largely superseded by the *logical IF* which makes a program easier to read and understand.

A *four*-way switch can be achieved using a *logical IF* statement incorporating an *arithmetic IF* statement:

```
      IF ( X+Y  .GT.  100.0)  IF (X+Y)  10, 20, 30
      I = N                OBEYED NEXT IF
                           (X+Y) ≤ 100.0
```

but this is the sort of "clever" programming best avoided.

ASSIGNED GO TO

"CASE C OF ..."
BUT IT IS BEST NOT TO USE IT

Fortran 66 provides an alternative to the *computed GO TO* mechanism. This involves special assignments of the form:

ASSIGN *label* **TO** *variable*

where:

label is the label of an executable statement in the same program unit as the *ASSIGN* statement. This item consists only of digits.

variable is the name of a variable (not an array element) of type *INTEGER*

This assignment is used in conjunction with a special *GO TO* statement of the form:

GO TO *variable* **,** *(label, label, ... , label)*

N.B.

where :

variable is the name of an integer variable to which a valid label has (when the *GO TO* is obeyed) already been assigned by an *ASSIGN* statement.

label is as defined for the *ASSIGN* statement. The parentheses surround a list of labels which must include that which has already been assigned to *variable* when the *GO TO* is obeyed.

Here is an example of the *assigned GO TO* in use:

```
      INTEGER     LABL
      ASSIGN  10  TØ  LABL
```

```
      ASSIGN  30  TØ  LABL
      IF ( I .EQ. J )  ASSIGN  20  TØ  LABL
```

LABELS OF EXECUTABLE STATEMENTS

```
      GØ  TØ  LABL , (10, 20, 30)
```

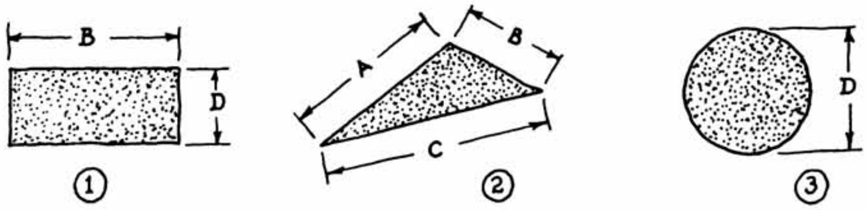
When the *ASSIGN* statement is obeyed the integer variable is assigned some "value" (a pattern of binary digits) corresponding to the label of a statement. *ASSIGN 10 TO LABL* is not the same thing as *LABL = 10*. So *LABL* should not be referred to save by *ASSIGN* statements. When *GO TO LABL* is obeyed control is transferred to the statement bearing the label last assigned to *LABL*. This *must* be one of those in the bracketed list.

Different Fortrans have different restrictions and extensions to the *assigned GO TO* mechanism so this form of control is best not used in programs intended to be portable. The *computed GO TO* described on page 39 provides a safer and better mechanism for dealing with the shape "*case C of ...*".

AREAS OF SHAPES

AN EXAMPLE TO ILLUSTRATE CONTROL STATEMENTS

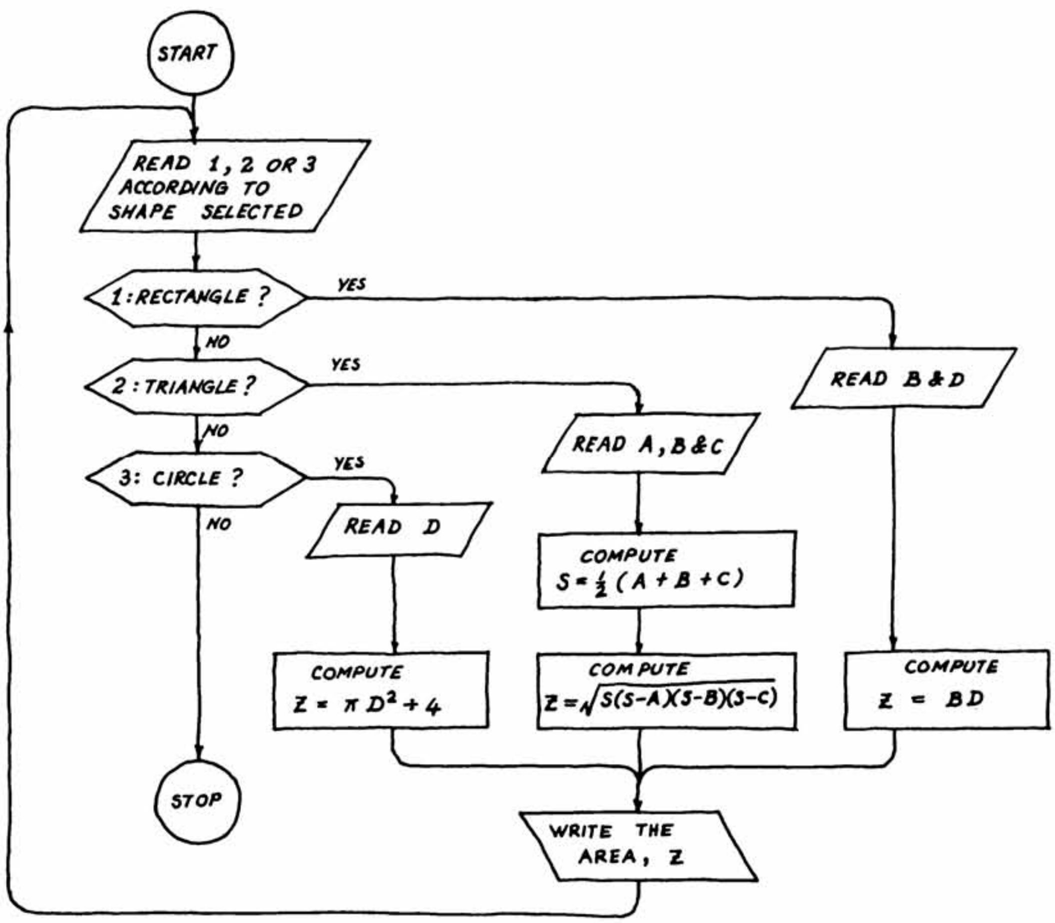
Here is a program designed to calculate areas of rectangles, triangles and circles.



The data comprise two lines (records) for each shape. The first line is given as 1 or 2 or 3 according to the shape required: the second line contains two or three or one dimension(s) appropriate to the shape selected. Thus the data could look like this:

COLUMN 1	COLUMN 10	COLUMN 20	COLUMN 30	
2				TRIANGLE
15.4	16.8	21.95		A, B & C
1				RECTANGLE
13.67	10.0			B & D
2				ANOTHER TRIANGLE
3.0	4.0	5.0		A, B & C
3				CIRCLE
15.9				D
0				END OF RUN

A flow chart for the program is shown below:



The program itself could be written thus:

```
      REAL    A, B, C, D, Z, S                 SPECIFICATION STATEMENTS
      INTEGER    ITYPE                         ARE DESIRABLE BUT
                                               NOT NECESSARY
   50 READ  (5, 100)   ITYPE
      IF  ((ITYPE .LT. 1)  .ØR.  (ITYPE .GT. 3))  STØP
      GØ  TØ   (10 , 20 , 30) ,  ITYPE                     "CASE
   10 READ  (5, 200)   B, D                                 C OF.."
      Z = B * D
      GØ  TØ   40
   20 READ  (5, 300)  A , B , C
      S  =  0.5 * ( A + B + C )
      Z  =   SQRT ( S *(S-A)* (S-B)*(S-C) )    SQRT( ) IS A FORTRAN 'FUNCTION' (CHAPTER 6)
      GØ  TØ   40
"WHILE"
   30 READ  (5, 400 )   D
      Z  =  3.141593 * D * D / 4.0
   40 CØNTINUE
      WRITE (6, 500)  Z
      GØ  TØ   50
  100 FØRMAT  (I1)                INTEGER IN FIRST COLUMN OF CARD
  200 FØRMAT  ( 2F10.0)
  300 FØRMAT  (3F10.0)
  400 FØRMAT  (F10.0)
  500 FØRMAT  (1X,  9H AREA IS  ,  F7.2 )
      END
```

Because there are only three possibilities specified in the statement *GO TO (10, 20, 30), ITYPE* , this line could be replaced by the old-fashioned *arithmetic IF* as follows:

```
      IF  ( ITYPE - 2)   10,  20,  30
```

Finally the output (using the data opposite) would look like this:

```
AREA IS   129.02
AREA IS   136.70
AREA IS     6.00
AREA IS   198.56
```

ALL the control statements introduced in this chapter – *viz*:

IF (logical expression) statement
GO TO label
STOP
GO TO (label, ...,label), variable
CONTINUE
DO label control = initial, terminal, increment
IF (arithmetic expression) label, label, label
ASSIGN label TO variable
GO TO variable, (label, ...,label)

are *executable* statements and may therefore carry labels.

EXERCISES

CHAPTER 4

4.1 A pair of simultaneous equations:

$$aX + bY = p$$
$$cX + dY = q$$

may be solved using Cramer's rule as follows:

$$X = \frac{\begin{vmatrix} p & b \\ q & d \end{vmatrix}}{\begin{vmatrix} a & b \\ c & d \end{vmatrix}} \quad \text{and} \quad Y = \frac{\begin{vmatrix} a & p \\ c & q \end{vmatrix}}{\begin{vmatrix} a & b \\ c & d \end{vmatrix}}$$

where the vertical bars denote *determinants* which may be evaluated as follows:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = a \times d - c \times d$$

Write a program to read the coefficients a, b, c, d and the right-hand side p, q then print the solution for X and Y. (Make sure you check the denominator is not zero before dividing: print the message "NO SOLUTION" if it is.) Then loop back to read a new right-hand side.

4.2 Write a program to generate and print the Fibonacci series:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

where each successive term is the sum of the previous two terms. (These numbers represent the numbers of rabbits generated by an initial pair of rabbits. The numbers grow rapidly because when each generation breeds so does its parents' generation, its grandparents' generation, its great grand... Computer programs for printing Fibonacci numbers must be proliferating even faster, because this boring example seems to appear in many text books on programming.) Make sure your program stops before the term next to be printed exceeds the capacity of an integer variable. One way to do this is to check that *half* the previous term, plus *half* the term before that, does not exceed *half* the capacity of a variable.

4.3 Write a program (just as boring) to evaluate and print the *factorial* of an integer read as data. As an example of a factorial, factorial five – written *5!* – is given by:

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Again make sure the capacity of an integer variable is not exceeded.

Alternatively, make the program print a *table* of factorials starting at *1!*

5

ARRAYS

TYPES OF ARRAY
SUBSCRIPTS
RIPPLE SORT (AN EXAMPLE)
EXERCISES

TYPES OF ARRAY

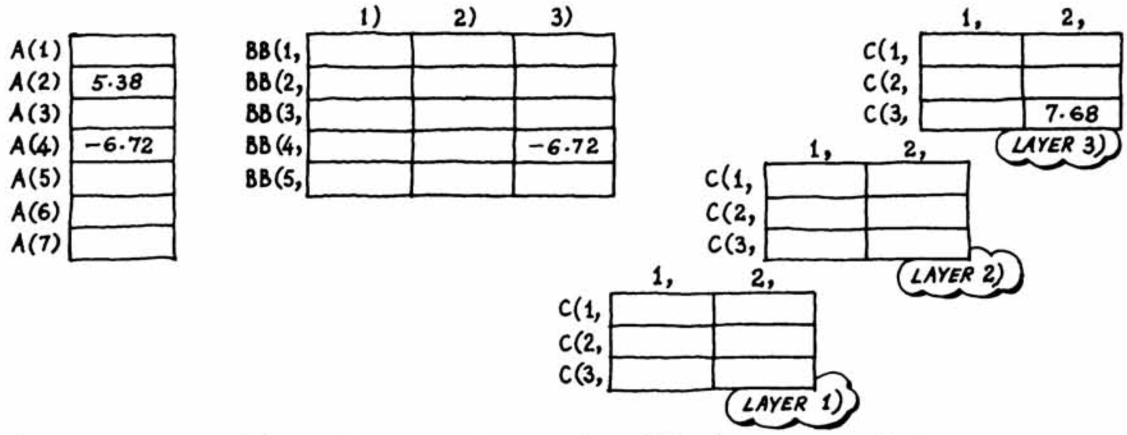
REAL, INTEGER, DOUBLE - PRECISION, LOGICAL, COMPLEX

An array is an arrangement of *elements* in one, two or three dimensions. The following:

```
REAL       A, BB, C
DIMENSIØN  A(7) ,  BB(5,3) , C(3,2,3)
```

ROWS COLUMNS LAYERS

specifies three arrays of type *REAL* depicted as follows:



Each array consists of a number of "little boxes" called *array elements*. Except where specifically mentioned to the contrary, each array element may be used in the manner of a *variable* of like type. The element is written as the name of the array followed by one, two or three *subscripts* in brackets:

```
A(2)  =  5.38
BB(4,3)  =  -6.72
C(3,2,3)  =  7.68
A(4)  =  BB(4,3)
```

SEE DIAGRAMS ABOVE

The *DIMENSION* statement has the form:

DIMENSION *array*, *array*, ... , *array*

where:

array is the symbolic name of an array followed by its dimensions in parentheses. For each array there may be one, two or three dimensions separated by commas. Except in the case of subprograms (Chapter 7) each dimension may conist only of digits.

The *DIMENSION* statement — being a specification statement — is non-executable and therefore should not be labelled. *DIMENSION* statements should follow the *type* stataments as defined by the table on page 17.

The *DIMENSION* statement may be omitted if arrays have their dimensions specified by *type* statements:

```
REAL     A(7) ,  BB(5,3) ,  C(3,2,3)
INTEGER    IA(3,2)
```

DIMENSION STATEMENTS OMITTED

As in the case of variables, if the type of an array is not specified it becomes *implicitly* specified by the initial letter of the array's name.

I , J , K , L , M , N

Arrays whose names begin with letters *I, J, K, L, M, N* are implicitly specified as type *INTEGER*: those with other initials as type *REAL*. So the previous two statements could be reduced to one:

```
      DIMENSION  A(7), BB(5,3), C(3,2,3), IA(3,2)
```

There is a third way of specifying the dimensions of an array – by the *COMMON* statement as illustrated below:

```
      INTEGER  IA
      REAL   A , BB , C
      COMMON  A(7) ,  BB(5,3) , C(3,2,3), IA(3,2)
```

where the use of *COMMON* is explained in Chapter 8.

It is wrong to specify dimensions more than once for any array:

```
      REAL   RED(6,6)
      DIMENSION   RED(6,6)
```

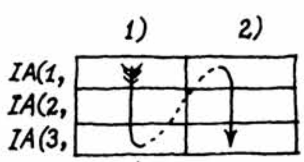
Arrays are stored *by columns*. For example, in the integer array:

```
      DIMENSION    IA(3,2)
```

elements are stored in the order:

IA(1,1), IA(2,1), IA(3,1), IA(1,2), IA(2,2), IA(3,2)

in other words:



THE FIRST SUBSCRIPT VARIES FASTEST

In many Fortrans no check is made on the bounds of an array; thus it is possible, for example, to write *IX = IA(4,1)* to pick up the value stored in IA(1,2). This is allowable in Fortran 66 but not in Fortran 77. It may even be possible, using some compilers, to pick up the same value by writing *IX = IA(4)*. For portable programs, however, such tricks should never be employed. The number and range of every subscript should conform to the dimensionality and size ranges originally specified.

Although some Fortrans set all elements of each array to zero at start of execution most do not. The content of every array element – just like the content of every variable – is *undefined* at start of execution (there is no saying *what* it might contain) unless pre-set by a *DATA* statement. *DATA* statements are described in Chapter 9.

SUBSCRIPTS

*ONLY SEVEN FORMS ARE PERMITTED (integer constant * integer variable ± integer constant)*

Each subscript of an array element may be written as an integer constant as already illustrated:

```
      A(2) = 5.38
      BB(4, 3) = -6.72
```

INTEGER CONSTANTS

A subscript may also be written (within limits) as an *expression* of type *INTEGER* :

```
      A(2)            (INTEGER CONSTANT)
      A(I)            (INTEGER VARIABLE†)
      A (I + 2)       (INTEGER VARIABLE† ± INTEGER CONSTANT)
      A (I - 2)
      A (2*K)         (INTEGER CONSTANT * INTEGER VARIABLE†)
      A (2*K + 1)     (INTEGER CONSTANT * INTEGER VARIABLE† ±
      A (2*K - 1)           INTEGER CONSTANT)
```

† NOT AN ARRAY ELEMENT

Although many Fortrans (including Fortran 77) permit more complicated expressions as subscripts there are only seven forms permitted by Fortran 66 ≈ and all seven forms are illustrated above. Even an array element written *A(1 + 2*K)* or *A(K*2)* is non-standard and likely to prevent a program being fully portable.

An array element may be used in the manner of a variable except where specifically noted. Thus it is wrong (page 40) to write:

```
      DØ 10 I = IA(1, 1) , IA(1, 2)
```

but the intended effect may easily be obtained at the cost of extra assignments:

```
      J = IA (1, 1)
      K = IA (1, 2)
      DØ 10 I = J , K
```

Here are a few examples of the use of array elements. First a piece of program to read values into a one-dimensional array (often called a *vector*). Each value is punched in the first ten columns of a card:

```
      REAL A(7)
      DØ 10 I = 1, 7
      READ (5, 100) A(I)
   10 CØNTINUE
  100 FØRMAT (F10.0)
```

THIS LOOP MAY BE REPLACED BY THE SINGLE STATEMENT: READ(5, 100) A (PAGE 96)

and here is a piece of program to "clear" a two-dimensional array:

```
      REAL BB(5, 3)
      DØ 20 ICØLUM = 1, 3
      DØ 20 IRØW = 1, 5
      BB(IRØW, ICØLUM) = 0.0
   20 CØNTINUE
```

Here is a piece of program to print a one-dimensional array (a vector) as a column of values down the left-hand side of the output page:

```
      DØ 30  I = 1, 7
      WRITE (6, 200)  A(I)
   30 CØNTINUE
  200 FØRMAT  (1X,  F10.2)
```

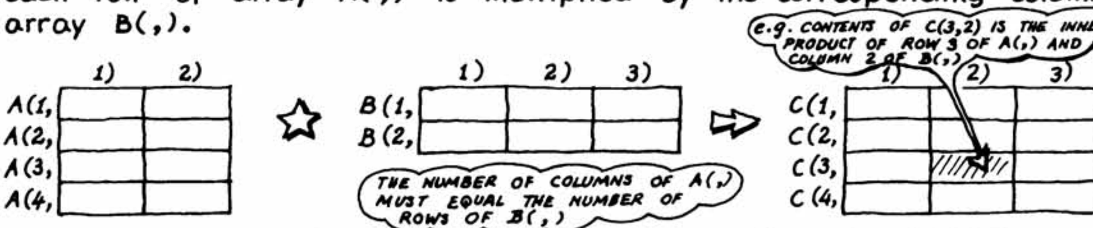
THIS LOOP MAY BE REPLACED BY THE SINGLE STATEMENT: WRITE (6,200)A (PAGE 96)

In the next piece of program a vector is *scanned* to find the location of the first negative value (if any):

```
      DØ 40  I = 1,7
      IF ( A(I) .LT. 0.0 ) GØ TØ 50
   40 CØNTINUE
      I = 0
   50 WRITE ( 6, 300)  I
  300 FØRMAT  ( 1X,  13H LØCATIØN IS , I2 )
```

NO NEGATIVE VALUE FOUND

The following example shows a *matrix multiplication*. Each element of array C(,) is made to contain the sum of the products obtained when each row of array A(,) is multiplied by the corresponding column of array B(,).




```
      DIMENSIØN  A(10,10) , B(10,10) , C(10,10)
      INTEGER  RØWSA , CØLSA , CØLSB
```

```
      RØWSA = 4
      CØLSA = 2
      CØLSB = 3
C
      DØ 10  J = 1, CØLSB
        DØ 10  I = 1, RØWSA
          C(I,J) = 0.0
   10 CØNTINUE
C
      DØ 20  K = 1, CØLSB
        DØ 20  J = 1, RØWSA
          DØ 20  I = 1, CØLSA
          C(J,K) = C(J,K) + A(J,I) * B(I,K)
   20 CØNTINUE
```

SET UP THE DIMENSIONS

CLEAR ARRAY C(,)

COMPUTE THE INNER PRODUCTS

The above piece of program could be made more elegant and efficient by omitting the "DO 10" loops altogether and replacing the three "DO 20" loops with:

```
      DØ 20  K = 1, CØLSB
        DØ 20  J = 1, RØWSA
        X = 0.0
          DØ 30  I = 1, CØLSA
   30     X = X + A(J,I) * B(I,K)
        C(J,K) = X
   20 CØNTINUE
```

CLEAR JUST ONE VARIABLE

but is it as clear to you ?

RIPPLE SORT

AN EXAMPLE TO ILLUSTRATE SUBSCRIPTED VARIABLES

Sorting numbers into ascending order is simple in concept but remarkably difficult to organize when there are large volumes of data. The example below uses one of the simplest techniques of all — the *ripple sort* — which is adequate for small volumes of data (a hundred or so numbers) stored as a vector.

A(1)	6.5
A(2)	13.9
A(3)	10.2
A(4)	4.6
A(5)	3.5

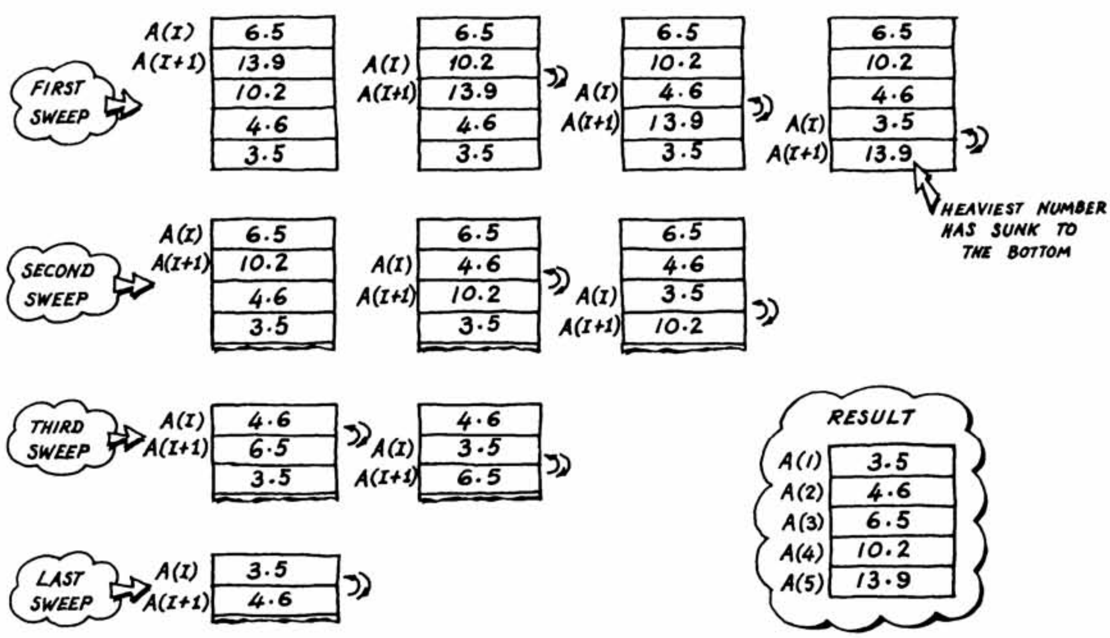
Array A() — a column vector — is to be sorted into ascending order; heaviest numbers sinking to the bottom. You can reverse this order by reversing the condition of the *logical IF* statement.

Start with an "index" *I* pointing to row 1; then advance *I* row by row. At every advance look at the number *I* is pointing to — and also at the number one row ahead of *I*. If the former is greater than the latter swop the two numbers.

Having finished one "sweep" of *I* sweep again — but stop one row short of the previous sweep because the heaviest number must already have sunk to the bottom.

Continue sweeping — each sweep a row shorter than the previous one — until there is a whole sweep without a single swop in it or the length of sweep is reduced to nothing.

Here is the whole process: ⟫ shows where a swop has just occurred:



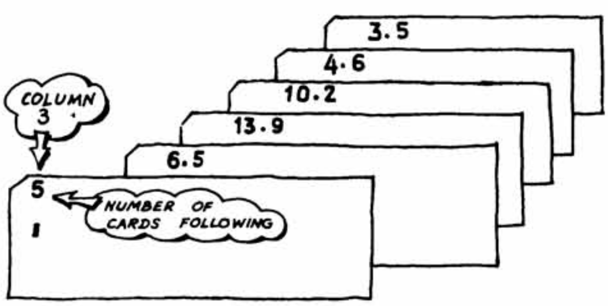
The program opposite is designed to sort a vector *A()*, having *N* rows. The vector is filled with numbers by the technique already illustrated; the results are printed down the left margin of the output page by the technique already illustrated.

Logical variable *EXIT* is set *true* at the start of each sweep but is changed to *false* if a swop has to be made. Thus if there is a full sweep without any swops, control jumps to label 40 for an early exit.

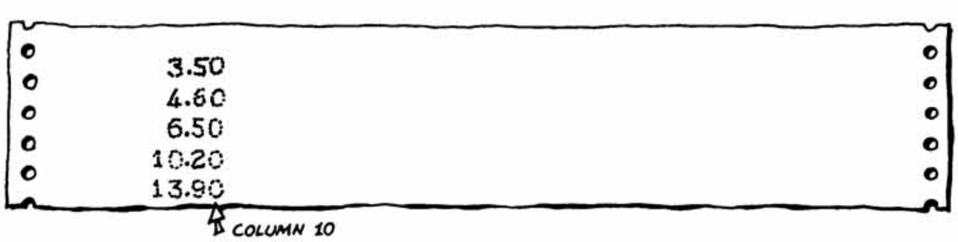
Here is the full program:

```
      REAL    A(100) ,  SWØP
      INTEGER    I ,  KARDS ,  NSWEEP ,  LAST , LENGTH
      LØGICAL        EXIT
C                                                   COUNT OF CARDS
      READ  (5, 100)  KARDS
      IF  ( (KARDS .LT. 2) .ØR. (KARDS .GT. 100) )  STØP 1
      DØ  10  I = 1, KARDS
      READ  (5,200)  A(I)                           READ EACH CARD
   10 CØNTINUE
C
      LAST = KARDS - 1
      DØ  20   NSWEEP = 1, LAST                     SWEEPS
      EXIT = .TRUE.
      LENGTH  = KARDS - NSWEEP                      SHORTER LENGTH EACH SWEEP
        DØ  30   I = 1, LENGTH
        IF ( A(I) .LE. A(I+1) ) GØ TØ  30
        SWØP  = A(I)
        A(I) = A(I+1)           SWOP &                    DON'T
        A(I+1) = SWØP           DENY EXIT                 SWOP
        EXIT = .FALSE.
   30   CØNTINUE
      IF (EXIT) GØ TØ  40
   20 CØNTINUE                  EARLY EXIT
C
   40 CØNTINUE
      DØ  50   I = 1, KARDS
      WRITE   (6, 300)   A(I)                       PRINT RESULTS & STOP
   50 CØNTINUE
      STØP
C
  100 FØRMAT  (I3)                                  ROUNDED TO TWO
  200 FØRMAT  (F10.0)                               DECIMAL PLACES
  300 FØRMAT  (1X, F10.2)
C
      END
```

The input (if on punched cards) might look like this:



The output on unit 6 would look like this:



EXERCISES

CHAPTER 5

5.1 Using the early examples of this chapter as models write a complete program for reading two arrays; performing a matrix multiplication; printing the result. Try the program on the following data:

SALES PERSON	SALES PRODUCTS 1)	2)	3)	4)
A(1,	5.	2.	0.	10.
A(2,	3.	5.	2.	5.
A(3,	20.	0.	0.	0.

NUMBERS OF ITEMS SOLD

ARRAY A(,)

SALES PRODUCT	PRICE 1)	COMMISSION 2)
B(1,	1.50	0.20
B(2,	2.80	0.40
B(3,	5.00	1.00
B(4,	2.00	0.50

PRICES & COMMISSIONS LIST

ARRAY B(,)

and the result should be the values in the following table:

SALES PERSON	SALES AMOUNT 1)	SALES COMMISSION 2)
C(1,	33.10	6.80
C(2,	38.50	7.10
C(3,	30.00	4.00

SUMS & COMMISSIONS EARNED

ARRAY C(,)

= ARRAY A(,) × ARRAY B(,)

MATRIX MULTIPLICATION

5.2 There is a modification of the ripple sort called the *shuttle sort* which is explained in many books on programming. The method requires just one pass down the vector – but every time a swop is made, the number shifted up one place must be compared with the number now above it – and so on until this item has risen as far as it should go. Write a program to perform the shuttle sort.

5.3 The enthusiastic reader may care to explore three further methods of sorting explained in many text books on programming. These three methods are:

- *the Shell sort*
- *the monkey-puzzle sort*
- *Quicksort*

of which the last two are mind-benders but worth the effort of study.

6

SIMPLE FUNCTIONS

INTRINSIC FUNCTIONS
BASIC EXTERNAL FUNCTIONS
STATEMENT FUNCTIONS
TRIANGLE (AN EXAMPLE)
ROUGH COMPARISON (AN EXAMPLE)
EXERCISES

INTRINSIC FUNCTIONS *"BUILT IN" TO FORTRAN*

The introductory example showed the line:

```
NPØTS = INT ( PØTS) + 1
```

where *INT( )* is a reference to a *function* capable of converting its *argument* (in parentheses) from a real value to an integer value by truncation. In other words all digits after the decimal point are thrown away. *INT(3.999)* yields the integer result *3* standing in place of *INT(3.999)*.

There are thirty-one intrinsic functions in Fortran 66. The term "intrinsic" implies these functions are usually *built in* (rather than *linked on*) to the compiler. This, in turn, implies that you should not use any of their thirty-one names for any other purposes besides those defined opposite. In particular you should not use any of these names for functions of your own devising such as *statement functions* which are explained later.

The form of a *function reference* (or *invocation*) is:

name (argument, argument, ... , argument)

where:

name is one of the thirty one names defined in the table opposite. The *type* of each function (*i.e.* the type of value you get back from the function) is indicated by the initial letter of the name of each function (see initials below).

argument is the name of a variable or array element, or is an expression of the type indicated by the initial letter of each symbolic argument in the table opposite:

- *D* signifies a function or argument of type *DOUBLE PRECISION*;
- *C* signifies a function or argument of type *COMPLEX*;
- *I & M* signify functions or arguments of type *INTEGER*;
- • other initials signify functions of type *REAL*.

A function reference may be used in the manner of a variable except where specifically forbidden. Thus it would be wrong (page 40) to write:

```
DØ  10   I = INT(A + B), INT (C*D)
```

but the intended effect could be had at the cost of extra assignments:

```
J  = INT(A + B)
K  = INT(C*D)
DØ  10   I = J, K
```

An argument of a function — whether intrinsic or otherwise — may be an expression of much complexity:

```
DESCR = SQRT(FLØAT(IB**2 - 4*IA*IC))/FLØAT(2*IA)
```

FUNCTION	DEFINITION
ABS(*A*) IABS(*I*) DABS(*D*)	The absolute (positive) value of an argument: *ABS(-3.5)* is 3.5 ; *ABS(+3.5)* is 3.5 ; *IABS(-3)* is 3 ; *DABS(-3.5D0)* is 3.5D0
AINT(*A*) INT(*A*) IDINT(*D*)	Truncation: the sign of the argument is applied to the largest integer less than the absolute value of that argument: *AINT(-3.9)* is -3.0 ; *INT(-3.9)* is -3 ; *IDINT(1.5D0)* is 1
AMOD(*A1*, *A2*) MOD(*I1*, *I2*)	Remainder when *A1* is divided by *A2* : $= A1 - \lvert A1 \div A2 \rvert \times A2$ where the vertical bars contain an integer whose magnitude does not exceed *A1÷A2* and whose sign is the same as the sign of *A1÷A2* : *AMOD(5.0,2.0)* is *1.0* ; *AMOD(-5.0, 2.0)* is *-1.0* (similarly for *I1* and *I2*).
AMAX0(*I1*, *I2*, ..., *In*) AMAX1(*A1*, *A2*, ..., *An*) MAX0(*I1*, *I2*, ..., *In*) MAX1(*A1*, *A2*, ..., *An*) DMAX1(*D1*, *D2*, ..., *Dn*)	The value of the largest argument: *AMAX1(-99.5, -16.1, 2.3)* is 2.3 *AMAX0(-99, -16, 2)* is 2.0 (*i.e. REAL*) There must be at least two arguments in each function.
AMIN0(*I1*, *I2*, ..., *In*) AMIN1(*A1*, *A2*, ..., *An*) MIN0(*I1*, *I2*, ..., *In*) MIN1(*A1*, *A2*, ..., *An*) DMIN1(*D1*, *D2*, ..., *Dn*)	The value of the smallest argument: *AMIN1(-99.5, -16.1, 2.3)* is -99.5 *AMIN0(-99, -16, 2)* is -99.0 (*i.e. REAL*)
FLOAT(*I*) IFIX(*A*)	Conversion from type *INTEGER* to type *REAL*. Conversion from type *REAL* to type *INTEGER* by truncation: the same effect as *INT(A)* above.
SIGN(*A1*, *A2*) ISIGN(*I1*, *I2*) DSIGN(*D1*. *D2*)	The sign of the second argument (which should *not* be zero) applied to the absolute value of the first argument. *SIGN(3.5, -1.2)* is -3.5 ; *SIGN(-3.5, -1.2)* is -3.5 ; (similarly for *ISIGN* and *DSIGN*).
DIM(*A1*, *A2*) IDIM(*I1*. *I2*)	The positive difference: *DIM(A1,A2)* is *A1 - AMIN1(A1, A2)* and *IDIM(I1,I2)* is *I1 - MIN0(I1, I2)*.
SNGL(*D*)	The most significant part of a double-precision argument expressed as a single-precision result — properly rounded.
DBL(*A*)	A single-precision argument extended to double precision.
REAL(*C*) AIMAG(*C*)	The real and imaginary parts of a complex argument respectively: *REAL((4.0, 2.0))* is 4.0 ; *AIMAG((4.0, 2.0))* is 2.0
CMPLX(*AR*, *AI*)	Express as a complex number with real part *AR* and imaginary part *AI* : *CMPLX(4.0, 2.0)* is (4.0, 2.0) (in other words $4.0 + 2.0 \times \sqrt{-1}$ — a complex number).
CONJ(*C*)	Obtain the conjugate (imaginary part reversed in sign) of a complex argument: *CONJ((4.0, 2.0))* is (4.0, -2.0) (in other words $4.0 - 2.0 \times \sqrt{-1}$).

BASIC EXTERNAL FUNCTIONS "OFFERED" BY FORTRAN

Consider this little program for printing a table of square roots:

```
      DØ 10   I = 1, 12
      A =  SQRT ( FLØAT (I))
      WRITE (6 , 100)  I , A
   10 CØNTINUE
      STØP
  100 FØRMAT ( 1X , 14HSQUARE RØØT ØF, I4 , 3H IS, F10.4)
      END
```

On the second line *FLOAT( )* is an *intrinsic* function: *SQRT( )* is a *basic external* function. Both kinds of function may be invoked in precisely the same way. The difference (theoretically) is that the programmer may *override* any basic external function by devising his own function (see later) and giving it the same name. However, because some Fortrans do not stick to the letter of the standard it is safer *not* to override basic external functions but treat them in the same way as intrinsic functions. Thus if you want to devise your own square-root function give it some name other than *SQRT* : for example *SQROOT*.

There are twenty-four basic external functions in Fortran 66 and these are tabulated opposite. Many Fortrans, however, offer additional ones. For example there is no *tangent* function defined by Fortran 66 but many Fortrans offer *TAN( )* as a basic external function. Using such non-standard functions invites problems of portability ≈ don't be tempted!

The form of a *function reference* (or *invocation*) is the same as that of an intrinsic function:

name (argument, argument, ... , argument)

where:

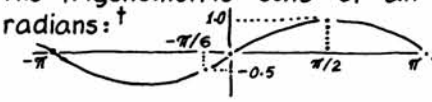
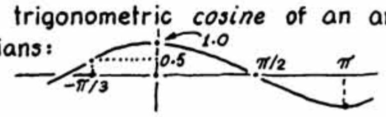
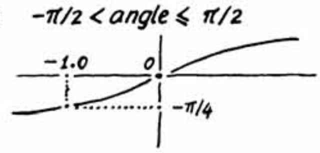
name is one of the twenty-four names defined in the table opposite. The *type* of each function (*i.e.* the type of value you get back from the function) is also tabulated because the initial letter of the function's name does not always indicate type.

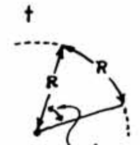
argument is the name of a variable or array element, or is an expression of the type indicated by the initial letter of the symbolic arguments in the table opposite:

- *A* signifies an argument of type *REAL* ;
- *D* signifies an argument of type *DOUBLE PRECISION*
- *C* signifies an argument of type *COMPLEX*

As in the case of intrinsic functions a *function reference* may be used in the manner of a variable except where specifically forbidden. Also the arguments may be complicated expressions, themselves containing function references. The following statement works out the fourth root of the absolute value of a *REAL* number stored in array element X(2) :

```
      R4 = SQRT( SQRT (ABS ( X(2))))
```

FUNCTION	TYPE OF FUNCTION	DEFINITION
EXP(*A*) DEXP(*D*) CEXP(*C*)	*REAL* *DOUBLE PRECISION* *COMPLEX*	e raised to the power given by the argument: in other words the *natural antilogarithm* of the argument. *EXP(0.0)* is *1.0* ; *EXP(1.0)* is e.*
ALOG(*A*) DLOG(*D*) CLOG(*C*)	*REAL* *DOUBLE PRECISION* *COMPLEX*	The *natural* (base e) logarithm of an argument which must have a value greater than zero: *ALOG(1.0)* is *0.0* ; *ALOG(e)* is *1.0*
ALOG10(*A*) DLOG10(*D*)	*REAL* *DOUBLE PRECISION*	The *common* (base 10) logarithm of an argument which must have a value greater than zero: $ALOG_{10}(10.0)$ is *1.0*
SIN(*A*) DSIN(*D*) CSIN(*C*)	*REAL* *DOUBLE PRECISION* *COMPLEX*	The trigonometric *sine* of an argument in radians:†  *PI = 3.14159...* *SIN(-PI/6.0)* is *-0.5* *SIN(0.0)* is *0.0* *SIN(PI/2.0)* is *1.0*
COS(*A*) DCOS(*D*) CCOS(*C*)	*REAL* *DOUBLE PRECISION* *COMPLEX*	The trigonometric *cosine* of an argument expressed in radians:  *PI = 3.14159...* *COS(-PI/3.0)* is *0.5* *COS(0.0)* is *1.0* *COS(PI/2.0)* is *0.0*
ATAN(*A*) DATAN(*D*)	*REAL* *DOUBLE PRECISION*	The arctangent of an argument = *the angle (in radians) whose tangent is...* The range of the result is $-\pi/2 < angle \leqslant \pi/2$  *ATAN(∞)* is $\pi/2$ *ATAN(0.0)* is *0.0* *ATAN(-1.0)* is $-\pi/4$
ATAN2(*A1, A2*) DATAN2(*D1, D2*)	*REAL* *DOUBLE PRECISION*	The arctangent of *A1 ÷ A2* but the signs of *A1* and *A2* are individually significant and *A2* may be zero (similarly for *D1* & *D2*). The range of the result is $-\pi < angle \leqslant \pi$
TANH(*A*)	*REAL*	The hyperbolic tangent of an argument: $(e^{A} - e^{-A}) \div (e^{A} + e^{-A})$ (*i.e. sinh(A)/cosh(A)*)
SQRT(*A*) DSQRT(*D*) CSQRT(*C*)	*REAL* *DOUBLE PRECISION* *COMPLEX*	The square root of an argument which may not have a negative value
DMOD(*D1, D2*)	*DOUBLE PRECISION* (*see also the intrinsic functions MOD(,) & AMOD(,)*)	The remainder when *D1* is divided by *D2*: $= D1 - \lvert D1 \div D2 \rvert \times D2$ where the vertical bars contain an integer whose magnitude does not exceed the magnitude of *D1 ÷ D2* and whose sign is the same as the sign of *D1 ÷ D2*.
CABS(*C*)	*REAL* (*see also the intrinsic functions ABS( ), IABS( ) & DABS( )*)	The *REAL* modulus of a complex argument. If *C* is *(AR, AI)* then *CABS(C)* is: *SQRT (AR**2 + AI**2)*

† 

1 radian (*π radians = 180°*)

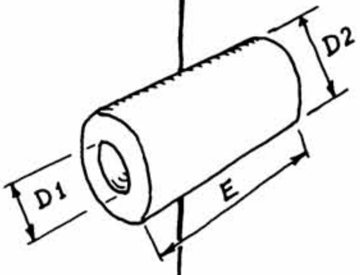
* $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$

STATEMENT FUNCTIONS

DEVISED BY THE PROGRAMMER

The following program is for computing the mass of a length of pipe, given its length, both diameters, and density of material: *E, D2, D1, RHO*.

```
      REAL  E, D2, D1, RHØ, A2, A1, W
      READ (5,100)  E, D2, D1, RHØ
      A1 = 3.141593 * D1 **2 / 4.0
      A2 = 3.141593 * D2 **2 / 4.0
      W  = E * (A2-A1) * RHØ
      WRITE (6,200)  W
      STØP
100   FØRMAT ( 4F 10.0)
200   FØRMAT ( 1X, 7HMASS IS, F10.2)
      END
```



The lines beginning *A1* and *A2* do much the same job and may be turned into a single *statement function*. The program could be recast thus:

```
      REAL  E, D2, D1, RHØ, W, X, AREA
      AREA(X) = 3.141593 * X **2 / 4.0
      READ (5,100)  E, D2, D1, RHØ
      W  =  E * ( AREA(D2) - AREA(D1) ) * RHØ
      WRITE (6,200) W
```

DEFINES A 'STATEMENT FUNCTION' CALLED AREA( )

FIRST EXECUTABLE STATEMENT

STATEMENT FUNCTION INVOKED TWICE

Notice the statement function is defined immediately before the first executable statement. The statement function may be invoked (some say "referenced") anywhere else within the program unit in which it is defined. But if one statement function invokes another the invoked one must be defined first:

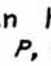
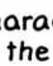
```
      AREA(X) = 3.141593 * X **2 / 4.0
      VØLUME (P, Q)  =  AREA(P) * Q
      WEIGHT (R, S, T)  =  VØLUME(R, S) * T
      READ  (5,100)  E, D2, D1, RHØ
      W   =   WEIGHT( D2, E, RHØ)-WEIGHT(D1,E,RHØ)
      WRITE (6,200) W
```

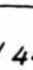
STATEMENT FUNCTIONS IN CORRECT ORDER

FIRST EXECUTABLE STATEMENT

'WEIGHT' FUNCTION INVOKED TWICE

The variable *X* in the first statement function is a *dummy argument* having no connection with any other *X* that might be used in the same program. We could just as well have defined the statement function:

AREA ( *) = 3.141593 *`*`*  *`**` 2 / 4.0*

except that Fortran has no such character as . The same goes for dummy variables *P, Q, R, S, T* in the piece of program above.

```
      AREA(X)  =  3.141593 * X **2 / 4.0
```

DUMMY ARGUMENT!

The form of definition of the statement function is shown below. This is not an executable statement and therefore should not be labelled.

name (dummy, dummy, ... , dummy) = expression

where:

name is the symbolic name chosen for the statement function. It should not be the name of an intrinsic function, and it is safer that it should not be the name of a basic external function or any of Fortran's keywords. It should not be the name of any variable or array in the same program unit. The name may be declared in a *type* statement, or the type of function becomes implicitly declared by the initial letter of *name*.

dummy is the symbolic name chosen for a dummy argument. The types of dummy arguments may be specified in preceding *type* statements, otherwise their types are implicitly specified by initial letters (*I* to *N* specify *INTEGER*; other initials *REAL*).

expression is an arithmetic expression or logical expression involving *all* the dummy arguments. Array elements are not allowed in this expression. The *type* of expression must agree with the type declared (explicitly or implicitly) for the function's name.

The statement function may be invoked in the manner of an intrinsic or basic external function as illustrated opposite. Here it is again:

```
      W = E *(AREA (D2) - AREA(D1) ) * RHØ
```

"ACTUAL" ARGUMENT   "ACTUAL" ARGUMENT

where the *actual argument* may be a complicated expression of appropriate type. But the actual argument may not be the name of an array, nor may it be an *EXTERNAL* name (page 70).

Fortran 66 has no *tangent* function although Fortran 77 does. It is simple to provide a *tangent* function in the form of a statement function:

```
      TAN(X) = SIN(X) / CØS(X)
```

and similarly for other trigonometric and hyperbolic functions missing from the definition of Fortran 66. For example:

```
      SINH(X) = (EXP(X) - EXP(-X)) / 2.0
```

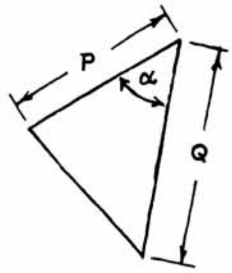
Fortran allows the *expression* in the definition to contain variables other than *dummy* variables, but this practice is not recommended because it may make a program difficult to understand and errors hard to trace. In the example below *A, B, C* and *D* are defined elsewhere in the program unit: their current values are used each time the statement function is invoked. *A, B, C, D* are called *parameters*.

```
      CUBIC(X) = A*X**3 + B*X**2 + C*X + D
```

TRIANGLE

AN EXAMPLE TO ILLUSTRATE INTRINSIC, BASIC EXTERNAL AND STATEMENT FUNCTIONS IN USE

Given the lengths of two sides of a triangle and the included angle:



it is possible to derive expressions for the area, the other two angles, and the length of the third side.

WHEN A & B ARE ACUTE ANGLES

area $= \frac{1}{2} PQ \sin\alpha$

angle A $= \tan^{-1}\left(\frac{Q \sin\alpha}{P - Q\cos\alpha}\right)$

angle B $= \tan^{-1}\left(\frac{P \sin\alpha}{Q - P\cos\alpha}\right)$

length of third side, $L = \sqrt{P^2 + Q^2 - 2PQ\cos\alpha}$

Here is a set of data for the program. The values represent the two lengths, P and Q, followed by the included angle, α, expressed in degrees:

17.5	20.0	45.0	

COLUMN 10 COLUMN 20 COLUMN 30

Here is a program to compute the area, angle A, angle B, and length L:

```
      REAL    L
      DEGREE(R) = R*180.0/3.141593
      RADIAN(D) = D*3.141593/180.0
C
      READ (5, 100)   P, Q, ALPHA
      C = RADIAN(ALPHA)
      AREA = 0.5 * P * Q * SIN(C)
      IA = INT(DEGREE(ATAN(Q*SIN(C)/(P-Q*CØS(C)))))
      IB = INT(DEGREE(ATAN(P*SIN(C)/(Q-P*CØS(C)))))
      L =  SQRT(P*P + Q*Q - 2.0*P*Q*CØS(C))
C
      WRITE (6, 200)  AREA, L
      WRITE (6, 300)  IA, IB
      STØP
  100 FØRMAT (3F10.0)
  200 FØRMAT (1X, 7HAREA IS, F10.2, 17H ØPPØSITE SIDE IS, F10.2)
  300 FØRMAT (1X, 11HBASE ANGLES , I6, 4H AND, I6)
      END
```

STATEMENT FUNCTIONS
degrees ← radians
radians ← degrees

YES! IT COULD BE MADE SIMPLER, BUT THIS PROGRAM WAS CONTRIVED TO ILLUSTRATE FUNCTIONS

and the output would appear as:

```
AREA IS    136.12 OPPOSITE SIDE IS     15.68
BASE ANGLES     62 AND     52
```

ROUGH COMPARISON

ILLUSTRATING A LOGICAL STATEMENT FUNCTION

Tests for approximate equality are rather messy in the body of a program. A statement function of type LOGICAL such as:

```
      LØGICAL  EQUAL
      EQUAL(X, Y, APPRØX) = ABS(X-Y) .LE. APPRØX
```

can make such a program tidier. This statement could be invoked as follows:

```
      RUFFLY = 0.001
      IF (.NØT. EQUAL(B*B, 4.0*A*C, RUFFLY))GØ TØ 10
```

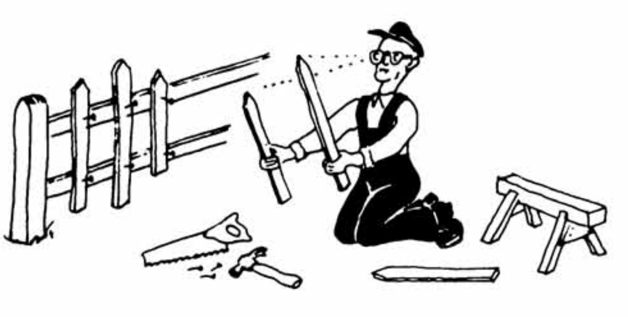
A quantity such as RUFFLY in the example above is better set in a DATA statement as defined in Chapter 9 but illustrated below:

```
      DATA    RUFFLY / 0.001 /
```

where it may be easily found by the programmer and its value changed if the criterion for convergence is to be altered.

The logical function could, of course, be invoked with the criterion for approximate equality written as a constant:

```
      IF ( EQUAL( P, Q, 0.001) ) STØP
```



EXERCISES

6.1 Write a program to compute and print the highest common factor of two integers read as data. Use Euclid's algorithm:

- divide the larger integer, L, by the smaller, S
- if there is no remainder then S is the answer
- otherwise copy the content of S into L and put the remainder into S
- go back to the first step.

This method requires the use of the intrinsic function *MOD(,)*

6.2 The sum of the first *N* integers is given by:

$$\frac{N(N+1)}{2}$$

Make a statement function called *ISUM(N)* to deliver this result.

6.3 The area of a triangle with sides of length *A, B, C* is given by $\sqrt{S(S-A)(S-B)(S-C)}$ where *S* is the semi-perimeter (half of *A+B+C*). Make a statement function called *AREA(, ,)* using the three lengths as dummy arguments. (As a statement function this will be long and messy. The function would be nicer written as a *function subprogram*. These are explained in the next chapter.)

6.4 Present the body of the "loans" program on page 31 as a statement function.

7

FUNCTION AND SUBROUTINE SUBPROGRAMS

FUNCTION SUBPROGRAMS
SUBROUTINE SUBPROGRAMS
EXTERNAL
HORRORS
AREAS OF POLYGONS (AN EXAMPLE)
EXERCISES

FUNCTION SUBPROGRAMS

FUNCTIONS DEVISED BY THE PROGRAMMER

The programming language called *BASIC* provides a useful function called *SGN( )* which returns a value of *+1.0* or *0.0* or *-1.0* depending upon whether the single argument proves to be positive or zero or negative respectively. A similar function may be included in a Fortran program by writing a *function subprogram* as follows:

```
      REAL FUNCTION SGN(X)
      IF (X .GT. 0.0) SGN = 1.0
      IF (X .EQ. 0.0) SGN = 0.0
      IF (X .LT. 0.0) SGN = -1.0
      RETURN
      END
```

A FUNCTION SUBPROGRAM

The function may then be invoked from another program unit as though it were an intrinsic function; actual arguments conforming in type and usage with corresponding dummy arguments:

```
      A = SGN(B*C) * SQRT(ABS(B*C))
```

MAIN PROGRAM

INVOCATION

ACTUAL ARGUMENT (REAL)

A function subprogram begins with a heading containing the word *FUNCTION* and ends with an *END line* to tell the Fortran compiler there are no more statements to compile. The form of the heading is:

FUNCTION *name (dummy, dummy, ..., dummy)*

or:

type **FUNCTION** *name (dummy, dummy, ..., dummy)*

where:

type may be **INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL** depending upon the type of result to be handed back to the program which invoked the function. If omitted (as in the first of the two forms defined above) the type is implicitly declared by the initial letter of *name* by the standard convention: **I** to **N** imply type *INTEGER*; other initials type *REAL*.

name is the symbolic name given to the function by the programmer. This name must not appear in any *type* statement within the function subprogram; the means of declaring type has just been described.

dummy is a symbolic name given to a dummy argument of any type. There must be at least one such argument in a function subprogram. Its type may be specified by a *type* statement in the subprogram or be implicitly declared by the initial letter of the dummy's name.

The dummy argument may represent a *variable* or an *array* or an *external subprogram* (all three forms are illustrated; the third on page 70).

Names of dummy arguments may not be included in *EQUIVALENCE* statements (page 80) or in *COMMON* statements (page 76) or in *DATA* statements (page 86) within the function subprogram.

Somewhere in the subprogram there must be at least one executable statement of the form:

RETURN

The *RETURN* statement makes control return to the program which invoked the function. Also there must be at least one *assignment* having the *name of the function* on the left of the equals sign. This assignment determines what value is handed back to the program which invoked the function.

The purpose of a function subprogram is to return a single value in its place ~ just like an intrinsic function. The *mechanics* of Fortran 66 allow the programmer to write a function subprogram that:

- alters the values of its arguments
- changes values in *COMMON* (page 76)

but for the sake of portability *these things should never be done*. This is the province of the *subroutine subprogram* described next.

Here is a different version of the *SGN()* function; it is of type *INTEGER* but has an argument of type *DOUBLE PRECISION*:

```
      INTEGER FUNCTIØN  IDSGN(X)
      DØUBLE  PRECISIØN  X
      IDSGN = 0
      IF (X .GT. 0.0D0)  IDSGN = 1
      IF (X .LT. 0.0D0)  IDSGN = -1
      RETURN
      END
```

DUMMY ARGUMENT AS A DOUBLE-PRECISION VARIABLE NAME

The following function is designed to add up six consecutive elements of a vector:

```
      REAL  FUNCTIØN   SUMSIX(VECTØR)
      DIMENSIØN   VECTØR(6)
      SUMSIX  =  0.0
      DØ  10   I = 1, 6
   10 SUMSIX  =  SUMSIX  + VECTØR(I)
      RETURN
      END
```

DUMMY ARGUMENT AS AN ARRAY NAME

and this function could be invoked from another program unit as follows:

```
      DIMENSIØN  X(6), Y(6)
      ...
      TØTAL = SUMSIX(X)  +  SUMSIX(Y)
```

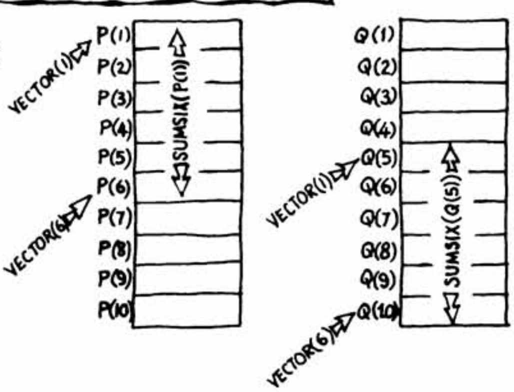
ARRAY NAMES — MAIN PROGRAM

or it could be invoked in a more subtle way by using the name of an *array element* as the actual argument rather than just the name of an array:

```
      DIMENSIØN P(10), Q(10)
      ...
      SUBTØT  =  SUMSIX(P(1))  +  SUMSIX(Q(5))
```

ARRAY ELEMENT NAMES — MAIN PROGRAM

Here the actual argument specifies the *first of six elements to be summed*. Notice that an actual argument of P(1) has the same effect as an actual argument of P, but that this is not true of Q(5) & Q.



It is the responsibility of the person who *invokes* such a function to ensure that the function works inside the bounds of the nominated array. Arrays with *adjustable dimensions* (these are permitted in functions) are described on page 69.

SUBROUTINE SUBPROGRAMS *PROGRAM UNITS WHICH ARE "CALLED"*

Here is an example of a *subroutine* designed to do the same job as that done by the *SGN( )* function illustrated earlier:

```
      SUBRØUTINE   SGNS (RESULT, EXPRSN)
      RESULT = 0.0
      IF (EXPRSN .LT. 0.0) RESULT = -1.0
      IF (EXPRSN .GT. 0.0) RESULT = +1.0
      RETURN
      END
```

This could be invoked (in the case of a subroutine it is usual to say "called") from another program unit as follows. The point of return is the first executable statement following the *CALL*; in this case at the line labelled *30*:

```
      CALL SGNS(ANSWER, B*C)
   30 WRITE (6, 100) ANSWER
```

RESULT RETURNED BY SUBROUTINE SGNS(,)

A subroutine subprogram has the same essential structure as a function subprogram. The form of the heading is:

SUBROUTINE *name*

or:

SUBROUTINE *name* (*dummy*, *dummy*, ... , *dummy*)

where:

name is the symbolic name given to the subroutine by the programmer. The initial letter has no significance.

dummy is a symbolic name given to a dummy argument of any type. The type may be declared by a *type* statement inside the subprogram or be implicitly declared by the initial letter of the dummy's name. Unlike a function subprogram a subroutine subprogram need not have any argument ~ hence the first of the two forms above.

A dummy argument may represent a *variable*, an *array* or an *external subprogram* (all three forms are illustrated; the third on page 70).

Names of dummy arguments may not be included in *EQUIVALENCE* statements (page 80) or in *COMMON* statements (page 76) or in *DATA* statements (page 86) within the subroutine subprogram.

Somewhere in the subroutine subprogram there must be at least one executable statement of the form:

RETURN

to make control return to the program unit from which the subroutine was called. The point of return is the first executable statement after the *CALL* statement as illustrated above.

The *CALL* statement is an executable statement of the form:

CALL *name*

or:

CALL *name* (*actual*, *actual*, ... , *actual*)

Where:

name is the symbolic name of the subroutine subprogram to be called

actual is an actual argument conforming in type and usage with the corresponding dummy argument of the subroutine subprogram being called.

The subroutine subprogram may communicate via its arguments as already illustrated. But a subroutine may also communicate by referring to named *COMMON* blocks and blank *COMMON*. There are illustrations of this kind of communication in Chapter 12.

When subroutine *SGNS(,)* (opposite) is called, the first actual argument cannot, of course, be a constant or expression; it should be the *name* of a variable or the *name* of an array element – in other words a "little box" in which to carry back the result. Such an argument we can call an *output* argument because it delivers output from the subroutine. Conversely the second actual argument we can call an *input* argument, and this may be a constant or expression as well as the name of a variable or array element:

VARIABLE NAME

```
      CALL  SGNS(ANSWER, 2.5*A*B)
      CALL  SGNS(VEC(2*I+J), 2.5*A*B)
```

ARRAY-ELEMENT NAME

MAY BE AN EXPRESSION

The following subroutine is designed to add a given value to successive elements of a vector. The number of successive elements is specified as one of the arguments. Notice how this number, as a dummy argument, is used as a *dimension* of a vector. This usage is called **ADJUSTABLE DIMENSIONS**:

```
      SUBRØUTINE  ADDIN(VECTØR, NUMBER, VALUE)
      DIMENSIØN   VECTØR(NUMBER)
      DØ 10  I = 1, NUMBER
   10 VECTØR(I) = VECTØR(I) + VALUE
      RETURN
      END
```

ADJUSTABLE DIMENSIONS

This subroutine may be called using the name of an array as the first actual argument:

```
      DIMENSIØN    V(10)

      CALL ADDIN(V, 4, -3.0)
```

MAIN PROGRAM

or may be called using the name of an array *element* as the first argument:

```
      CALL ADDIN(V(3), 7, 4.0 )
```

and it is up to the person who writes *the program that calls the subroutine* to ensure the subroutine does not work outside the bounds of the nominated array.

V(1)	-3.0
V(2)	-3.0
V(3)	-3.0 +4.0
V(4)	-3.0 +4.0
V(5)	+4.0
V(6)	+4.0
V(7)	+4.0
V(8)	+4.0
V(9)	+4.0
V(10)	

In the above subroutine the first argument is an output argument and must therefore be a *name*. The other two arguments are input arguments and may therefore be constants (as in the examples) or expressions of appropriate type.

The use and abuse of arguments is further discussed on page 71.

EXTERNAL

SUBPROGRAMS WHOSE NAMES ARE USED AS ARGUMENTS OF OTHER SUBPROGRAMS (SKIP THIS ON FIRST READING)

Here are two function subprograms designed to return the *cosecant* and *secant* of an angle expressed in radians:

```
      REAL FUNCTIØN   CØSEC(X)
      IF (SIN(X) .EQ. 0.0 )  STØP 1
      CØSEC = 1.0/ SIN(X)
      RETURN
      END
```

BASIC EXTERNAL FUNCTION

```
      REAL FUNCTIØN   SEC(X)
      IF ( CØS(X) .EQ. 0.0)  STØP 2
      SEC = 1.0/ CØS(X)
      RETURN
      END
```

BASIC EXTERNAL FUNCTION

And here is a function subprogram with one dummy argument representing a *function subprogram* and the other representing an angle expressed in *degrees*:

```
      REAL  FUNCTIØN   TRIG(RATIØ , ANGLE)
      REAL  ANGLE
      TRIG = RATIØ (ANGLE * 3.141593/180.0)
      RETURN
      END
```

DUMMY ARGUMENT AS THE NAME OF A FUNCTION

The TRIG(,) function could be invoked from another program unit as illustrated below. The EXTERNAL statement tells the Fortran compiler that COSEC and SEC are not names of variables but *names of external subprograms*.

```
      EXTERNAL   CØSEC , SEC
```

ANGLE A IN DEGREES

```
      RHØ = C * TRIG(CØSEC, A) * TRIG(SEC, A)**2
```

$\rho = C \times \mathrm{cosec}(\alpha) \times \sec^2(\alpha)$

ACTUAL ARGUMENTS AS NAMES OF EXTERNAL FUNCTIONS

A fundamental principle of Fortran is that any program unit may be *separately* compiled. If you consider the two invocations of TRIG above being compiled – with the Fortran compiler knowing nothing about subprograms called COSEC or SEC – you will see the need for the preceding EXTERNAL statement. How else could the compiler know that COSEC and SEC were not just names of variables?

The EXTERNAL statement has the form:

EXTERNAL *name, name, ... , name*

where:

name is the name of an external subprogram (function or subroutine) used in the current program unit as an argument of another function or subroutine.

Names of *statement* functions may not be declared EXTERNAL or be passed as arguments. The same applies to *intrinsic* functions. But although Fortran 66 allows *basic external* functions to be declared EXTERNAL – and their names to be used as arguments like COSEC and SEC above – it is safer not to do so. Not all Fortrans agree what functions are intrinsic and what external.

The EXTERNAL statement is not an executable statement so should not be labelled.

HORRORS

THE USE AND ABUSE OF LOCAL VARIABLES AND ARGUMENTS (SKIP THIS ON FIRST READING)

In Fortran 66 all local variables in a subprogram (those not in *COMMON* storage ≈ page 76) become *undefined* as soon as the *RETURN* statement is obeyed. In other words a subprogram cannot remember what its local variables and local arrays contained from one invocation to the next. In the following example:

```
FUNCTIØN   SUMATE(X)
IF ( X .EQ. 0.0)  TØTAL = 0.0
TØTAL  =  TØTAL  +  X
SUMATE  =  TØTAL
RETURN
END
```

INITIALIZATION

'TØTAL' IS A LOCAL VARIABLE FORGOTTEN UPON RETURN

successive invocations such as:

```
A = SUMATE(0.0)  +  SUMATE(50.0) + SUMATE(25.0)
```

INITIALIZE

might well *not* cause a value of *0.0 + 50.0 + 25.0 = 75.0* to be assigned to variable *A*. In many Fortrans these values would be retained, but reliance on such a feature makes a program non-portable.

A subprogram cannot be expected to return values via its arguments if one actual argument becomes associated with another. Consider:

```
FUNCTIØN   NØRTY(J,K)
J  =  K + 1
NØRTY  =  K
RETURN
END
```

OUTPUT ARGUMENT

INPUT ARGUMENT

which is perfectly reasonable if invoked using two distinct actual arguments:

```
LAST = 2
NØW  =  NØRTY(NEXT, LAST)
```

which would result in *NOW* becoming 2 and *NEXT* becoming 3. But if *NORTY* were invoked as follows:

```
NØ  = 2
IDØUBT  =  NØRTY (NØ, NØ)
```

what would be the value assigned to *IDOUBT* ≈ 2 or 3 ? And what would be contained in variable *NO* ? Different Fortrans would probably deliver different values.

Unintentional associations cause obscure bugs in Fortran programs. Be particularly careful about associating a dummy argument with an entity in *COMMON* storage (page 78).

A subroutine subprogram (not a function subprogram) may, according to Fortran 66, have an actual argument which is a Hollerith constant (*e.g. 4HCARD*) corresponding to a dummy input argument. However, the use of this feature is bound to make a program non-portable because the number of characters that can be stored in a variable differs from one computer to another and from one Fortran to another.

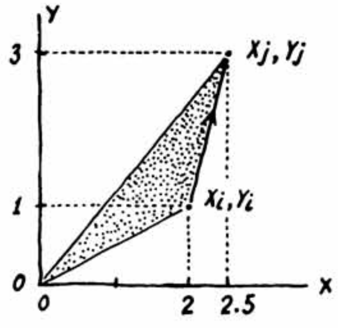
```
CALL    THINGY( I )
CALL    THINGY(4HCARD)
```

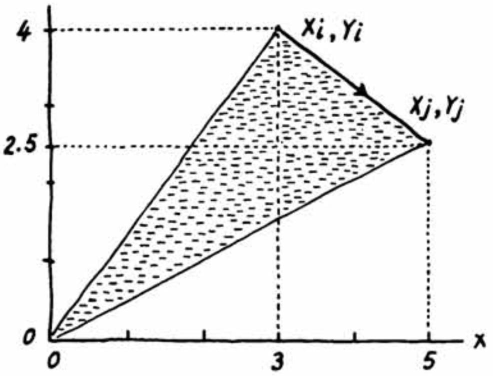
AREAS OF POLYGONS

ILLUSTRATING A FUNCTION SUBPROGRAM AND ITS INVOCATION

Consider the diagram on the right. It is a simple matter to show† that the speckled area is given by:

$$A_{ij} = \tfrac{1}{2}(X_i Y_j - X_j Y_i)$$
$$= \tfrac{1}{2}(2\times 3 - 2.5\times 1)$$
$$= 1.75$$

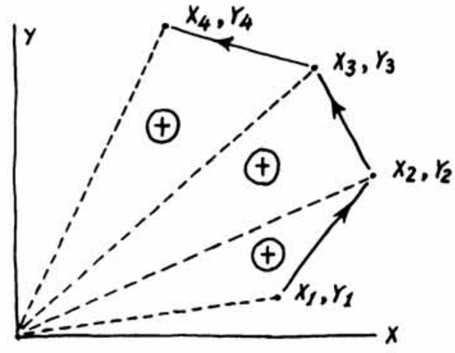


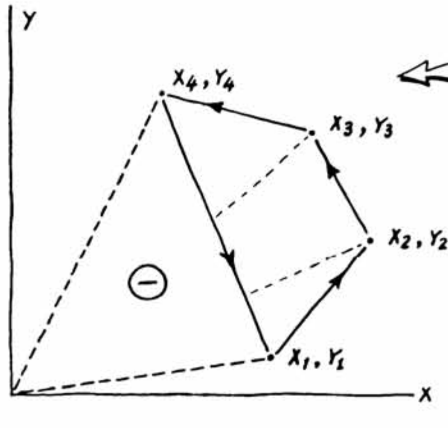


The same formula may be used for computing the area on the left. But this area would turn out to be *negative*:

$$A_{ij} = \tfrac{1}{2}(X_i Y_i - X_j Y_i)$$
$$= \tfrac{1}{2}(3\times 2.5 - 5\times 4)$$
$$= -6.25$$

The formula may be applied to sequential sides of a polygon ≈ and the areas of triangles summed to give the area shown on the right.



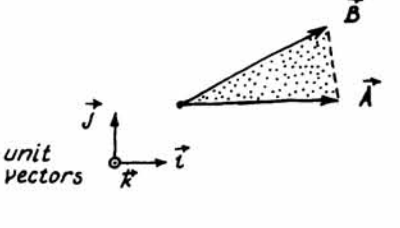


But if the polygon is *closed*, as shown on the left, the sum of all the triangular areas will equal the area of the polygon.

The bounded surface must be kept to the *left* of each arrow. No check is made on silly data where one edge crosses another as in a figure of eight. All points should have positive coordinates.

† the area enclosed between the two vectors $\vec{A}$ and $\vec{B}$ is given by $\tfrac{1}{2}(\vec{A}\times\vec{B})$, where

$$\vec{A}\times\vec{B} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \end{vmatrix} = (a_1 b_2 - b_1 a_2)\vec{k}$$



Here is a function subprogram devised to return the area of a polygon of *N* sides, where the coordinates of sequential vertices are stored in vectors *X( )* and *Y( )* :

```
      REAL FUNCTIØN AREA(X, Y, N)
      DIMENSIØN   X(N), Y(N)          ADJUSTABLE DIMENSIONS
      AREA = 0.0
      DØ 10  I = 1, N
             J = I + 1                CLOSES THE POLYGON BY
      IF (I .EQ. N) J = 1             JOINING FINAL VERTEX TO
                                      INITIAL VERTEX
      AREA = AREA + 0.5 * (X(I)*Y(J) - X(J)*Y(I))
   10 CØNTINUE
      RETURN
      END
```

A program which invokes this function could be written as follows:

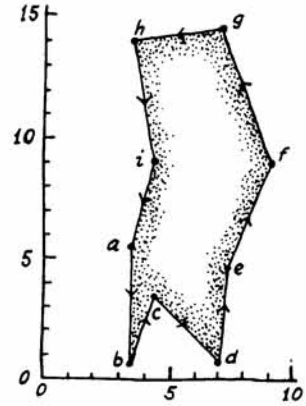
```
      DIMENSIØN   P(50), Q(50)
      READ  (5, 100)   NUMBER
      IF ((NUMBER .GT. 50) .ØR. (NUMBER .LT. 3)) STØP 1
      DØ 10   I = 1,   NUMBER
      READ  (5, 200)   P(I), Q(I)     INPUTS THE DATA
   10 CØNTINUE
C
C
      A = AREA(P, Q, NUMBER)          INVOKES THE FUNCTION
      WRITE  (6, 300)   A
      STØP
C
C
  100 FØRMAT  (I4)
  200 FØRMAT  (2F10.0)
  300 FØRMAT  (8H AREA IS, F10.1)
      END
```

Notice that variable *I* and label *10* in the subprogram have nothing whatever to do with variable *I* and label *10* in the main program. Communication is via function name and arguments only.

A set of data might be:

```
   9
     3.5            0.75
     4.25           3.5
     7.0            0.75
     7.25           4.75
     9.0            9.0
     7.0           14.5
     3.5           14.0
     4.25           9.0
     3.5            5.5
```

COLUMN 4 — COLUMN 10 — COLUMN 20



And the result would look like this:

```
 AREA IS       50.0
```

EXERCISES

CHAPTER 7

7.1 The "Macaulay function" *AMAC(X)* returns the value represented by its argument, *X*, if this value turns out to be positive; otherwise the function returns a value of zero. Write *AMAC( )* as a function subprogram.

7.2 Work through exercises 6.2, 6.3, 6.4 but this time write function subprograms instead of statement functions.

7.3 Recast the ripple-sort program on page 52 as a subroutine subprogram *RPSORT(A, N)* in which vector *A( )*, with adjustable dimension *N*, is sorted. Write a main program to call this subroutine. (See footnote.)

7.4 Rework exercise 5.1 so that the matrix multiplication is by a call to subroutine *MATMUL(A, B, C, I, J, K)* in which array *A(I,J)* is multiplied by array *B(J,K)* to give array *C(I,K)*. Write this subroutine and a main program to test it. (See footnote.)

7.5 Write a "library" of subroutines to perform the fundamental operations of matrix arithmetic:

- copying from one array to another: *MATCOP(A, B, I, J)*
- addition and subtraction: *MATADD(ISIGN, A, B, C, I, J)*
- scalar multiplication: *MATSCL(FACTOR, A, B, I, J)*
- transposition: *MATTRA (A, B, I, J)*
- clearing a matrix: *MATZER (A, I, J)*
- creating an identity matrix: *MATIDN(A, I)*
- matrix multiplication (as exercise 7.4 above)
- matrix inversion: *MATINV(A, B, I)*

For an introduction to computing with matrices see:
"Illustrating BASIC", Donald Alcock, Cambridge University Press, 1977.

Subroutine *MATIDN(A,I)* is done for you below, but you may want to devise a more elegant solution:

```
      SUBROUTINE    MATIDN (A, I)
      DIMENSION  A(I, I)
      DO 10   K = 1, I
      DO 10   J = 1, I
      A(J, K) = 0.0
      IF ( J .EQ. K)  A(J, K) = 1.0
   10 CONTINUE
      RETURN
      END
```

REPLACE ZERO DIAGONAL ELEMENT WITH 1.0

	1)	2)	3)
A(1,	1.0	0.0	0.0
A(2,	0.0	1.0	0.0
A(3,	0.0	0.0	1.0

footnote: you may wish to add an argument of type *LOGICAL* which is set *true* if the subroutine succeeds, or *false* if the actual arguments specify an impossible situation such as a negative number of elements.

8

COMMON STORAGE

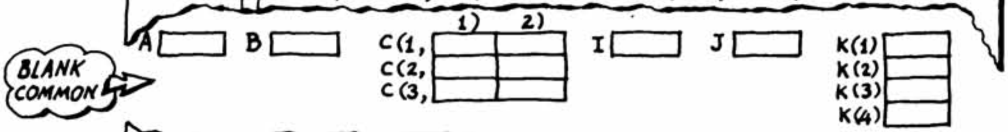
COMMON
COMMON (CONTINUED)
STACKS (AN EXAMPLE)
EQUIVALENCE
CHAINS (AN EXAMPLE)
EXERCISES

COMMON

A MEANS OF COMMUNICATION BETWEEN PROGRAM UNITS VIA SHARED VARIABLES AND ARRAYS

The *COMMON* statement declares names of variables and arrays that may be used by any or all program units in a complete program. Consider the following main program:

```
      CØMMØN  A, B,  C(3,2) , I, J, K(4)
```

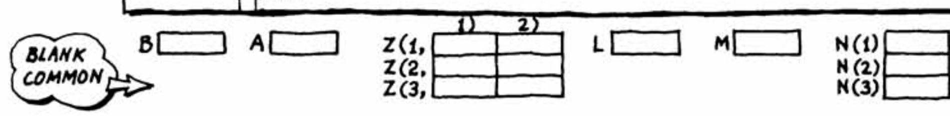



```
      CALL  NEGATE

      END
```

These variables and arrays may be referred to by the following sub-program:

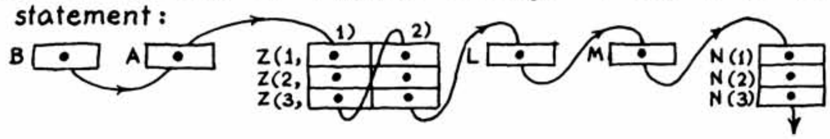
```
      SUBRØUTINE    NEGATE
      CØMMØN   B, A,  Z(3,2) , L , M , N(3)
      N(1) = -N(1)
      N(2) = -N(2)
      N(3) = -N(3)
      RETURN
      END
```



The elements of common storage declared in both program units bear a one-to-one correspondence ⇔ despite the completely different names being used. What the main program calls J the subprogram calls M, but it is nevertheless the same storage location in the computer. Notice that what the main program calls A the subprogram calls B; and what the main program calls B the subprogram calls A.

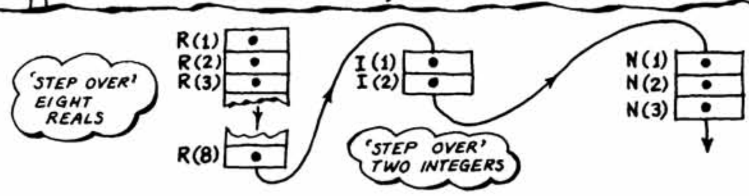
The effect of this subroutine subprogram when called is to negate the first three elements of array *K( )* on behalf of the main program.

The order of elements in common storage is the order declared in the *COMMON* statement:



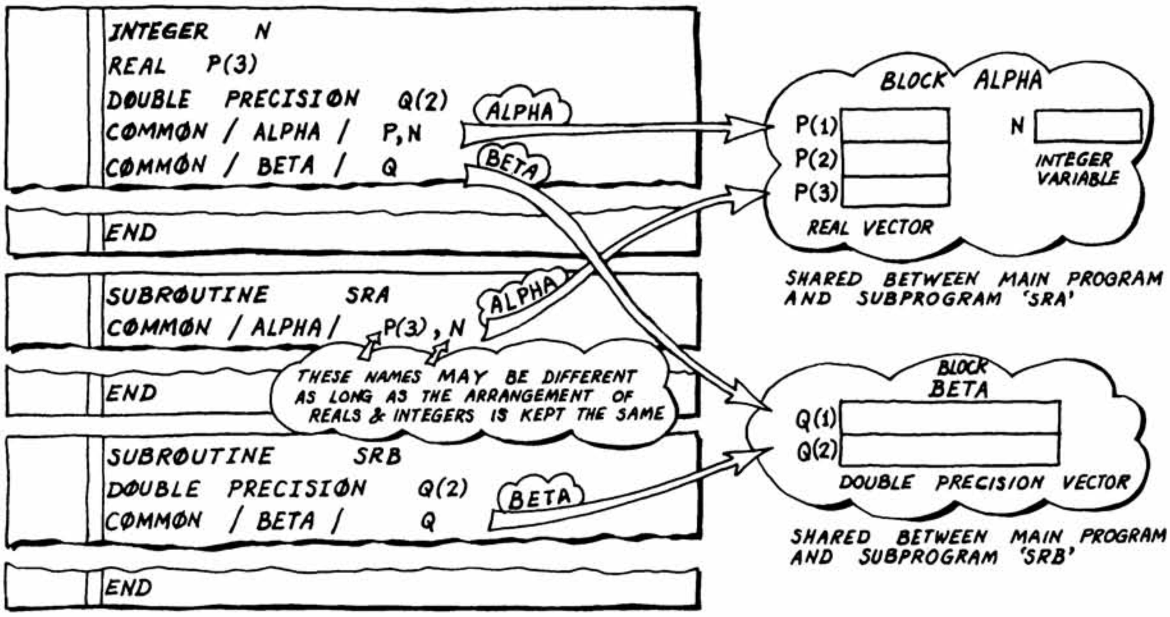
and because the subroutine above is only concerned with the array N( ) at the very end of common storage it would be allowable to simplify the common statement so as to "step over" unwanted elements as below:

```
      SUBRØUTINE    NEGATE
      CØMMØN    R(8) , I(2) ,    N(3)
```



Some pitfalls await the programmer who assumes anything about the relative sizes of elements having different type. These potential pitfalls are discussed later.

The opposite page illustrates *blank* (or *unlabelled*) COMMON. There is also *named* (or *labelled*) COMMON as depicted below:



Any number of named blocks may be specified. Each is accessible to any program unit that declares the block's name correctly and specifies a block of *identical length*. (Blank common need not be made the same length in every program unit: different lengths are illustrated opposite.)

The form of the COMMON statement is:

COMMON *name, name, ... ,name* ⟸ BLANK COMMON

or:

COMMON */block/ name, name, ... ,name*

where:

block is the symbolic name ≈ unique over all program units ≈ used to name a common block. This block may be referred to by any other program unit that declares a common block of the same name and size.

Omission of a block name (leaving a pair of slashes with nothing between them) specifies *blank* COMMON. In such a case the slashes may be omitted also, thus achieving the first of the two forms defined above.

name is the symbolic name of a variable of any type or the name of an array of any type. If it is the name of an array the name itself may be followed by the dimensions of that array in parentheses ≈ provided that these dimensions are not also given in a *type* or DIMENSION statement.

Fortran 66 allows more than one block to be declared in a single COMMON statement, but this can be confusing especially in the use of commas:

```
COMMON / ALPHA / P, N / BETA / Q
```
⟸ O.K. BUT CONFUSING

Dummy arguments may not be declared COMMON, nor may an array in common be given adjustable dimensions:

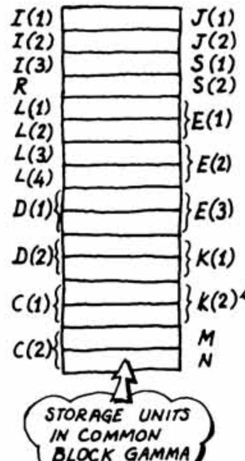
```
SUBROUTINE   WRONG(A, I, J)
DIMENSION   A(I, J)
COMMON     A
```

(continued)

Fortran 66 refers to *storage units* of which *one* is occupied by a variable or array element of type *INTEGER*, *REAL* or *LOGICAL* and of which *two* are occupied by a variable or array element of type *DOUBLE PRECISION* or *COMPLEX*. So ideally it should be simple to map one common block upon another; in other words to *associate* items one with another.

```
      INTEGER I(3)
      REAL R
      LOGICAL L(4)
      DOUBLE PRECISION D(2)
      COMPLEX C(2)
      COMMON / GAMMA / I, R, L, D, C
```

```
      INTEGER J(2)
      REAL S(2)
      DOUBLE PRECISION E(3)
      COMPLEX K(2)
      LOGICAL M, N
      COMMON / GAMMA / J, S, E, K, M, N
```

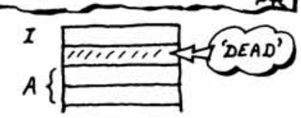


But there are few Fortrans in which this ideal would apply. It would be remarkable if complex elements C(1) and K(2) found themselves in the same pair of storage units.

In reality the fundamental storage unit is the computer *word* which has a different length on different computers. It may be 8, 16, 24, 32, 36, 48 or 60 binary digits (bits) in length. On a typical computer having a 16-bit word an integer or logical variable occupies one word, a real variable occupies two, a double precision or complex variable occupies four words. The diagram above would look quite different if drawn with a word as a storage unit.

To make things more complicated, some compilers "map" multi-word items onto words of store in such a way that "dead" words are left in the common block – making association difficult if not impossible.

```
      COMMON / DELTA / I, A
```




```
      LOGICAL L
      COMMON / THETA / L(3)
```

```
      LOGICAL I, J, K
      COMMON / THETA / I, J, K
```

And in some Fortrans, logical *arrays* are "packed" into successive words using only one bit per element, whilst each logical *variable* occupies a whole word. It is therefore impossible to associate logical variables with logical array elements.

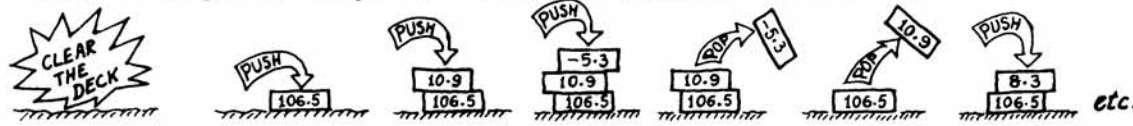
But it *is* possible to use blank and named *COMMON* in portable programs. The rules are these:

- always declare common variables and arrays in the order: *DOUBLE PRECISION* then *COMPLEX* then *REAL* then *INTEGER* then *LOGICAL*
- never associate items of different type
- never associate logical variables with logical arrays
- to be quite safe from losing data in named common blocks (because of overlay problems on some computers) include the definition of named blocks in the main program if any of their initial values are to be changed during execution. (See Bibliography – second book – on this subtle point.)

STACKS

AN EXAMPLE TO ILLUSTRATE THE USE OF A NAMED COMMON BLOCK

Stacks are widely used in programming. The technique has its own vocabulary involving the word "push" (to mean adding a number to a stack) and "pop" (to mean taking a number off a stack). The technique — used in anger on page 129 — may be depicted as follows:



The information pushed and popped may include integers representing labels of statements, letters in a word, arithmetic symbols — in fact information of almost any kind. The picture above illustrates a stack of real numbers.

Here are three subroutines for maintaining a stack of real numbers in a common block named *STACK*. The first subroutine should be called to clear the deck, but this subroutine could be replaced by a *BLOCK DATA subprogram* (page 87) to initialize the value of *IPOINT*.

```
      SUBROUTINE  CLEAR
      CØMMØN / STACK / A(10), IPØINT
      IPØINT = 0
      RETURN
      END
```

The following subroutine is for pushing a real number onto the stack:

```
      SUBRØUTINE   PUSH ( EXPRN)
      CØMMØN / STACK / STK(10) , IPT
      IF ( IPT .GE. 10 )  STØP  10
      IPT = IPT + 1
      STK ( IPT) = EXPRN
      RETURN
      END
```

OVER-FULL STACK CAUSES ERROR STOP

And the following subroutine is for returning the value popped from the stack:

```
      SUBRØUTINE   PØP( TØP, ØK)
      LØGICAL   ØK
      CØMMØN / STACK / S(10) , IPNT
      ØK = IPNT .GT. 0
      IF (.NØT. ØK) RETURN
      TØP = S( IPNT)
      IPNT = IPNT - 1
      RETURN
      END
```

EMPTY STACK MAKES 'OK' RETURN .FALSE.

All three routines refer to a common block named *STACK*. Names of variables in each subroutine have been made different so as to emphasize the *independence* of names — and the total *dependence on order* — within a common block.

Typical invocations of these subroutines are:

```
      LØGICAL  ANY
```

```
      CALL  CLEAR
      CALL  PUSH( A/B**2)
```

INITIALIZATION BEFORE FIRST PUSH

```
      CALL  PØP(VALU, ANY)
      IF (.NØT. ANY)  STØP
```

'VALU' FROM TOP OF STACK

EQUIVALENCE

A MEANS OF SHARING STORAGE SPACE, BUT NOT FOR MAKING SYNONYMS

Consider the following function subprogram contrived to return the value of y in the cubic equation $y = ax^3 + bx^2 + cx + d$ given the value of x and the four constants:

```
      REAL FUNCTIØN  CUBIC(X, A, B, C, D)
      R = A * X **3
      S = R +  B * X**2
      T = S +  C * X
      U = T + D
      CUBIC = U
      RETURN
      END
```

The intermediate variables *R, S, T, U* may be made to share one and the same storage location by inserting an *EQUIVALENCE* statement in the function subprogram as follows:

```
      REAL FUNCTIØN  CUBIC(X, A, B, C, D)
      EQUIVALENCE   (R, S, T, U )
```

The form of the *EQUIVALENCE* statement is:

EQUIVALENCE (*list*), (*list*), ... , (*list*)

where:

list is a list of two or more items separated by commas. Each item in the list may be the name of a variable or array element, but *not* a dummy argument. In the case of an array element the subscripts must be integers, and there must be as many subscripts as the array has dimensions.

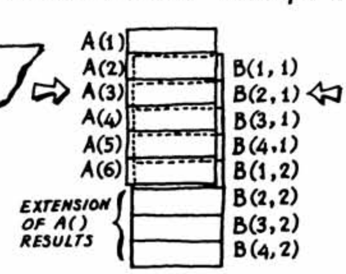
Despite the apparent simplicity of the *EQUIVALENCE* statement there are difficulties not always appreciated by Fortran programmers ~ difficulties that can easily make a program non-portable. Consider:

```
      REAL  D, E, F
      EQUIVALENCE   (D, E)
      D = 6.0
      E = 5.0
      F = D
```

Because *F* and *D* are of identical type, in most Fortrans the statement *F=D* would cause a value of *5.0* to be assigned to *F*. But Fortran 66 says that *D* (hence also *F*) would be *undefined*. Assignment to a variable or array element *undefines its equivalenced room-mates*. In other words the *EQUIVALENCE* statement does *not* imply synonyms. This one ghastly fact is enough to make this statement a danger in programs intended to be portable.

When two array elements are equivalenced the other elements in each array fall into place as depicted below. Two- and three-dimensional arrays are strung out by columns:

```
      REAL   A(6) ,   B(4,2)
      EQUIVALENCE   ( A(3) ,  B(2, 1) )
```



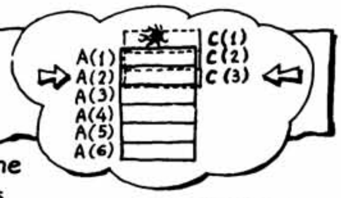
By the device just described it is permissible (but not good practice) to extend *COMMON* in a forward direction:

```
REAL  A(6) ,  B(4,2)
CØMMØN  A
EQUIVALENCE  ( A(3) , B(2,1) )
```

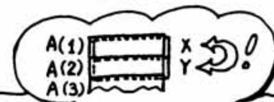
SEE SKETCH ON OPPOSITE PAGE

but it is not admissible to extend *COMMON* "backwards" (you can't create an *A(0)* or *A(-1)*):

```
REAL  A(6) ,  C(3)
CØMMØN  A
EQUIVALENCE  ( A(2) , C(3))
```



and it is obviously silly to equivalence one element of an array with another. This would imply "folding" the array over itself. This mistake could be made indirectly:

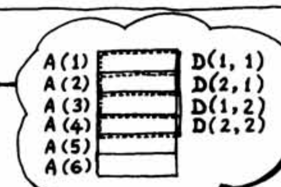



```
EQUIVALENCE (X,A(1)), (Y,A(2)), (X,Y)
```

Fortran 66 permits a single subscript of *unity* (no other) to imply the first element of a two- or three-dimensional array. Fortran 77, however, does not allow any such device:

```
DIMENSIØN  A(6) ,  D(2, 2)
EQUIVALENCE  ( A(1) , D(1) )
```

'77 bug



You can achieve what was intended above with greater clarity, and without falling foul of Fortran 77, as follows:

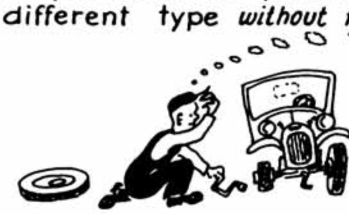
```
EQUIVALENCE  (A(1) , D(1,1) )
```

or even by equivalencing, say, *A(3)* with *D(1,2)*.

Fortran 66 allows items of different type to be equivalenced. For example the arrays *R(,)* and *I( )* may share space as a result of the following *EQUIVALENCE* statement:

```
REAL  R(3, 2)
INTEGER  I(12)
EQUIVALENCE  ( R(1,1) , I(1) )
```

Assuming a single computer word is used to store an integer, and a pair of words to store a real, arrays *R(,)* and *I()* might occupy the same space. In one well-tried and much-used program the programmer had taken advantage of this relationship, as a result of which the program would run on computers in which integers are stored in single words and reals in double words. On moving this program to a new computer ≈ in which an integer occupied the same space as a real ≈ the portability problem was overcome by the drastic means of turning all real arrays and variables into double-precision arrays and variables. To avoid problems of portability, therefore, do not equivalence items of different type *without forethought*.

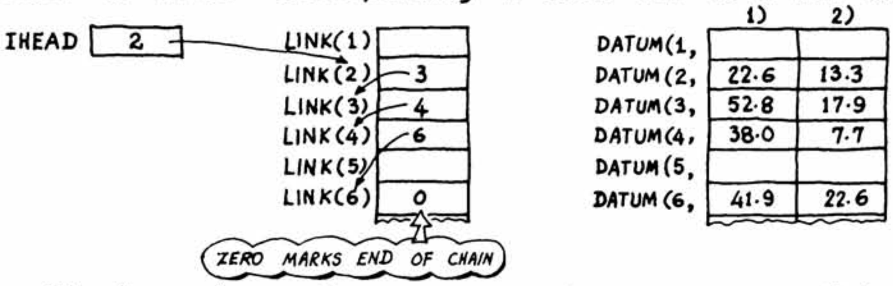


CHAINS

ILLUSTRATING A PROGRAMMING TECHNIQUE CALLED LIST PROCESSING

The manipulation of chains is called *list processing* ≈ a fundamental tool of programming. Chains are essential for solving the problem on page 126.

The simplest form of chain is illustrated below. It has a *head* and a vector of *links*. Corresponding to each link is a row of data:



The following piece of program would cause data linked by such a chain to be printed:

```
      N = IHEAD
   10 IF  (N .EQ. 0)  STØP
      WRITE  (6, 200)   DATUM(N,1), DATUM(N,2)
C
      N = LINK(N)
C
      GØ  TØ  10
  200 FØRMAT  (1X,  2F10.2)
```

END OF CHAIN

STANDARD SHAPE FOR CLIMBING DOWN TO NEXT LINK OF CHAIN

```
o          22.60          13.30          o
o          52.80          17.90          o
o          38.00           7.70          o
o          41.90          22.60          o
```

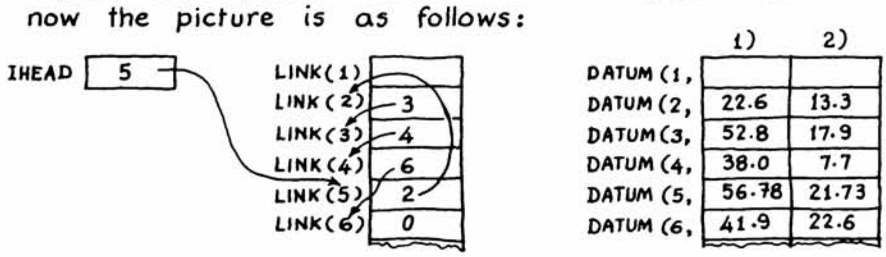
Suppose there were now some data in row 5 of array DATUM(,). The new data could be linked to the head of the chain by the following piece of program (in which variable *I* is first set to the desired row number):

```
      I = 5
      DATUM(I,1) =  56.78
      DATUM(I,2) =  21.73
C
      LINK(I) = IHEAD
      IHEAD = I
C
```

NEW DATA TO ROW 5

STANDARD SHAPE FOR LINKING ROW I TO HEAD OF CHAIN

and now the picture is as follows:



The last row linked may be *unlinked* (discarded) as shown below. (For the sake of illustration the row of data is printed before being lost.)

```
      WRITE  (6, 200)  DATUM(IHEAD, 1), DATUM(IHEAD, 2)
C
      IHEAD  =  LINK (IHEAD)
C
```

STANDARD SHAPE FOR DISCARDING TOP LINK

The output would be:

```
56.78            21.73      DISCARDED DATA
```

and the picture of arrays would once again be the first picture opposite (the one with *IHEAD* containing 2) but with "garbage" in row 5 of LINK( ) and of DATUM(,).

Notice these pieces of program cause the *last* row linked to be the *first* discarded – like pushing and popping a stack. Indeed programmers often use this chaining mechanism to organize stacks.

When using chains in more complicated ways it is useful to incorporate the head of each chain into the vector of links. This wastes a little space in the associated array of data, but there become fewer special cases to deal with at the heads of chains.

HEAD

	LINK	HEAD		1)	2)
LINK(1)	2		DATUM(1,	WASTED	
LINK(2)	3		DATUM(2,	22.6	13.3
LINK(3)	4		DATUM(3,	52.8	17.9

As an example, suppose we want a subroutine for discarding a certain link part way down a chain. The link to be discarded is the one linking a given value (*VALU*) to be found somewhere in the first column of array *DATUM(,)*. First of all this value has to be found by running down the chain and peeping one link ahead. If the value cannot be found then the subroutine returns a logical argument set *false*.

```
      SUBROUTINE   DSCARD ( VALU, ØK)
      LØGICAL   ØK
      CØMMØN / CHAIN / DATUM(100, 2) , LINK(100)
C
      NØW = 1                                  HEAD STORED IN LINK(1)
   10 NEXT = LINK(NØW)                         PEEP AHEAD
      ØK = NEXT .NE. 0
      IF (.NØT. ØK) RETURN                     'VALU' NOT FOUND: OK false
      IF ( DATUM (NEXT, 1) .EQ. VALU) GØ TØ 20   FOUND
      NØW = LINK (NØW)                         CLIMB DOWN
      GØ TØ 10
```

Then if the value is found the link may be discarded.

```
   20 LINK (NØW) = LINK (NEXT)
      RETURN                                   WITH OK true
      END
```

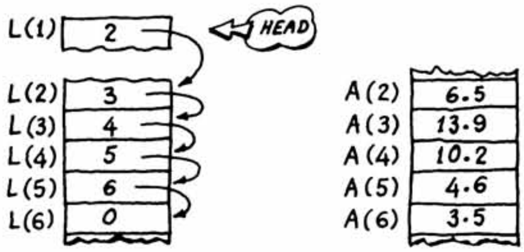
If you were to write this with the head of the chain in *IHEAD* you would have to make a special test for an empty chain before starting the search for *VALU*.

List processing is extremely useful, and, to enthusiasts, addictive. There is a good introduction in *Fortran Techniques* by Day (see Bibliography). This explains how to deal with "garbage" such as that left behind by the above subroutine every time a link is discarded.

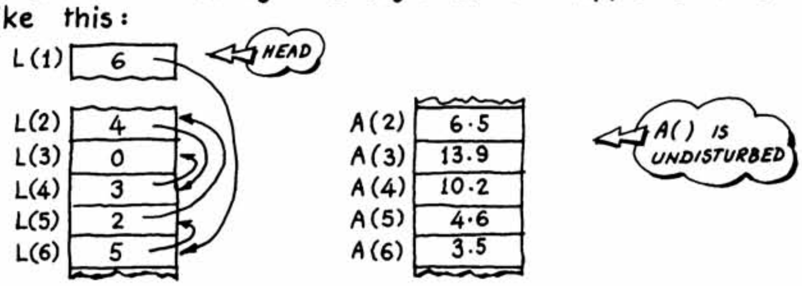
EXERCISES

8.1 Alter the "library" of subroutines specified in Exercise 7.5 so that each subroutine communicates via a named block of *COMMON* rather than having names of arrays as dummy arguments.

8.2 This is a challenge. Write a program to perform the ripple sort on numbers in vector *A( )* and print them in sorted order. But do not disturb vector *A( )*. Instead link the elements of *A( )* by a chain in *L( )*:



then *move the links* using the logic of the ripple sort so as to end up like this:



8.3 Turn your program (above) into a subroutine that will sort on any column of a two-dimensional array:

```
      CALL  MYSORT ( A, I, J, KEY)
```

where *I* and *J* specify the dimensions of array *A(I,J)* and *KEY* specifies which of the columns is to be used as the key for sorting. Test the subroutine by sorting on each column of a three-column array of numbers. Print the complete array ≈ in the order determined by the chain ≈ after each sort.

Hint: put the vector of links into a named common block. If you write a subroutine to do the initial linking:

```
      CALL  LINKER (LENGTH)        'LENGTH' ≡ 'I' ABOVE
```

you will only need one call to *LINKER* ≈ before the first call to *MYSORT*. Subsequent calls to *MYSORT* would then begin with the chain left behind after the previous call to *MYSORT*.

9

INITIALIZATION

DATA
BLOCK DATA
CHARACTERS
STATE TABLE (AN EXAMPLE)
EXERCISES

DATA A STATEMENT FOR INITIALIZING VARIABLES AND ARRAY ELEMENTS

The *DATA* statement is for setting initial values into local variables *before execution starts*. This is not an executable statement so:

- the *DATA* statement should not be labelled
- variables cannot be *reset* by a *DATA* statement during execution.

Here is a subroutine designed to return the surface area and volume of a sphere given its radius as an input argument:

```
      SUBROUTINE  SPHERE(RADIUS, AREA, VOLUME)
      DATA        PI / 3.141593 /
      AREA = 4.0 * PI * RADIUS **2
      VOLUME = 4.0 * PI * RADIUS**3/ 3.0
      RETURN
      END
```

LOCAL VARIABLE 'PI' IS INITIALIZED TO 3.141593 BEFORE EXECUTION – *i.e.* AT 'COMPILATION TIME'

More than one value may be set by a *DATA* statement. Here is a function to return the number of days in a month – given the month and year:

```
      INTEGER FUNCTION   DAYS(MONTH, YEAR)
      INTEGER  MONTH, YEAR, M(12), FEB, LEAP
      DATA  M(1), M(2), M(3), M(4), M(5), M(6), M(7), M(8), M(9), M(10)
     1    / 31 , 28 , 31 , 30 , 31 , 30 ,   2 * 31 , 30 , 31 /,
     2 M(11), M(12) / 30 , 31 /, FEB , LEAP / 2 , 4 /
      DAYS = M(MONTH)
      IF((MONTH .EQ. FEB) .AND. (MOD(YEAR/LEAP) .EQ. 0)) DAYS=29
      RETURN
      END
```

2*31 MEANS 31 , 31

The form of the *DATA* statement is:

DATA *names/constants/ , names/constants/ , ... , names/constants/*

where:

names is a list of names of variables or array elements or both. Items are separated by commas. Items may not be dummy arguments, nor may subscripts of array elements consist of anything but digits.

constants is a list of constants (Hollerith constants allowed) separated by commas. Arithmetic constants may be signed + or –. Any constant† may be preceded by a "multiplier" of the form:

count *

where *count* consists only of digits and specifies the number of times the subsequent constant is implied.

There must be a one-to-one correspondence between names in the *names* list and constants in the *constants* list. Each item in the *names* list must agree in type (*INTEGER, REAL, LOGICAL etc.*) with the corresponding item in the *constants* list. This agreement does not apply to Hollerith constants which may, according to Fortran 66, be stored in variables or array elements of *any* type. This topic is covered later in detail.

Many Fortrans allow you to write just the name of an array instead of laboriously listing its elements in sequence. For example the three-line *DATA* statement above is acceptable as follows:

```
      DATA  M/ 31, 28, 31, 30, 31, 30, 2*31, 30, 31, 30, 31/, FEB, LEAP/2, 4/
```

but this is, nevertheless, *non*-standard Fortran 66.

† *some Fortrans exclude Hollerith constants here.*

BLOCK DATA

A SUBPROGRAM FOR INITIALIZING VARIABLES AND ARRAY ELEMENTS IN NAMED COMMON BLOCKS

The illustrations opposite show the initialization of variables and array elements *local to the program unit* in which the *DATA* statements appear. Fortran doesn't allow items in *blank COMMON* to be initialized under any circumstances. Items in *named COMMON blocks*, however, may be initialized ≈ *but only in a BLOCK DATA subprogram*.

```
BLØCK DATA
REAL    R, RAR
INTEGER    I, IAR
DØUBLE PRECISIØN    D, DAR
LØGICAL    L, LAR
CØMPLEX    C, CAR
DIMENSIØN  RAR(100), IAR(100), DAR(100), LAR(100), CAR(100)
CØMMØN /IØTA/ R, RAR, I, IAR
CØMMØN /KAPPA/ D, DAR, C, CAR, L, LAR
EQUIVALENCE (IAR(1), J), (IAR(2), K), (IAR(3), M)
DATA  J, K, M /0, 1, 0/, RAR(1), RAR(2) /100.0, 0.0/
DATA  D, L, C / 1.5D0, .TRUE., (2.7, 3.6) /
END
```

A BLOCK DATA SUBPROGRAM

ONE ITEM MAY NOT BE IN MORE THAN ONE BLOCK

A *BLOCK DATA* subprogram begins with the statement:

BLOCK DATA

and ends with an *END* line:

END

between which can come any number of *type* statements (*REAL*, *INTEGER*, *DOUBLE PRECISION*, *LOGICAL*, *COMPLEX*) and any number of *DIMENSION* statements, *COMMON* statements, *EQUIVALENCE* statements and *DATA* statements. This order should be preserved; see page 17. No other kind of statement is allowed: the example above illustrates *all* these non-executable statements.

Fortran 77 (and several other Fortrans) permit any number of *BLOCK DATA* subprograms ≈ each having a name ≈ plus one unnamed subprogram like the one above. Portable programs, however, should have no more than *one unnamed BLOCK DATA subprogram*. This subprogram may be used to initialize any number of named *COMMON* blocks: the above *BLOCK DATA* subprogram initializes variables in *COMMON* blocks *IOTA* and *KAPPA*.

It is important to specify the *whole* of a named *COMMON* block ≈ even if you want to initialize just a few of its variables. Each named *COMMON* block must be seen to have precisely the same length in every program unit in which it is declared: the *BLOCK DATA* subprogam affords no exception.

The following *BLOCK DATA* subprogram could be used to replace the subroutine named *CLEAR* on page 79:

```
BLØCK DATA
CØMMØN / STACK /  A(10), IPØINT
DATA    IPØINT / 0 /
END
```

thus making it possible for the main program to do without the initialization statement: *CALL CLEAR*. However, some programmers consider it a mark of bad style to initialize in this way any variable that might later be changed in value.

CHARACTERS

INITIALIZATION IN A BLOCK DATA SUBPROGRAM
~ INTRODUCING "FREE FORMAT" INPUT ~

According to Fortran 66 (not Fortran 77 which has a special type) characters may be stored in variables or array elements of *any* type.

```
      REAL      R
      INTEGER     I
      DOUBLE   PRECISION    D
      LOGICAL    L
      COMPLEX      C
      DATA     R, I, D, L, C / 5 * 3HABC /
```

but this practice leads to non-portable programs because different computers have *words* of different length. On a typical 16-bit computer the effect of the above *DATA* statement might be as depicted below. The ▨ indicates a *blank* or *space* character.

R [A|B|C|▨]   *I* [A|B]   *D* [A|B|C|▨▨▨▨▨]   *L* [A|B]   *C* [A|B|C|▨▨▨▨▨]

The safest technique for portable programs is to store just one character per integer variable, or per element of an integer array. Here is a *BLOCK DATA* subprogram to initialize the Fortran 66 character set in a *COMMON* block named *SIGMA*:

```
      BLOCK   DATA
      COMMON   / SIGMA /   K(47)
      DATA  K(1), K(2), K(3), K(4), K(5), K(6), K(7), K(8), K(9), K(10)
     1  / 1H0 , 1H1 , 1H2 , 1H3 , 1H4 , 1H5, 1H6, 1H7, 1H8, 1H9 /
      DATA  K(11), K(12), K(13), K(14), K(15), K(16), K(17), K(18), K(19), K(20)
     2  /  1HA , 1HB, 1HC , 1HD, 1HE , 1HF , 1HG , 1HH , 1HI, 1HJ/
      DATA  K(21), K(22), K(23), K(24), K(25), K(26), K(27), K(28), K(29), K(30)
     3  /  1HK , 1HL, 1HM, 1HN, 1HO , 1HP, 1HQ , 1HR , 1HS, 1HT /
      DATA  K(31), K(32), K(33), K(34), K(35), K(36), K(37)
     4  /  1HU , 1HV , 1HW , 1HX , 1HY , 1HZ , 1H  /
      DATA  K(38), K(39), K(40), K(41), K(42), K(43), K(44), K(45), K(46), K(47)
     5  /  1H= , 1H+ , 1H- , 1H* , 1H/ , 1H( , 1H) , 1H, , 1H. , 1H$ /
      END
```

SPACE OR BLANK

Whenever a character is input it may be "looked up" in vector *K()*, *and its subscript used for any necessary manipulation*. There is a problem, however, in the "looking up". Comparison of characters is not trivial. Consider integer variables *I* [B|▨] and *J* [W|▨]. Because the characters occupy the most significant ends of *I* and *J* both variables hold *enormous* values. Furthermore the left-most bit is often used as a *sign* bit, so each integer value might be enormously positive or enormously negative. A statement involving *IF(I .EQ. J)...* might cause the computer to subtract one value from the other in order to test for a zero result. If the sign bits were originally opposed this operation might result in "integer overflow" and stop the program. The moral is to test the sign of each value and compare values only if the signs prove the same. Here is a logical function (used several times on subsequent pages) for discovering if two integer variables or array elements hold the same character or not:

```
      LOGICAL   FUNCTION   SAME (I, J)
      SAME  =  .FALSE.
      IF((I .LT. 0  .AND.  J.GE.0) .OR. (J .LT. 0  .AND.  I .GE. 0)) RETURN
      SAME  =  I  .EQ.  J
      RETURN
      END
```

RETURN false

true IF I & J ARE THE SAME, OTHERWISE false

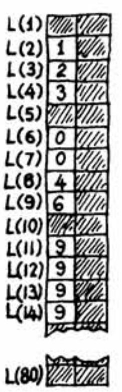
Using the subprograms opposite, here is a function to return a number (an "index" in the range 1 to 47) indicating what character occupies an integer variable. If this function returns zero it means the character occupying variable *N* is not in the Fortran 66 character set:

```
      INTEGER FUNCTIØN   INDEX(N)
      LØGICAL   SAME
      CØMMØN / SIGMA / K(47)          SHARES COMMON BLOCK 'SIGMA' SHOWN OPPOSITE
      DØ 10   I = 1, 47
      INDEX = I
      IF (SAME (N, K(I))) RETURN
   10 CØNTINUE                        INVOKES FUNCTION OPPOSITE
      INDEX = 0
      RETURN          CHARACTER NOT FOUND
      END
```

To illustrate a simple use of function *INDEX( )* here is a program to read a punched card (or line of data typed at a terminal) and print the values of unsigned integers punched into the card. The only characters allowed are *digits* and *spaces*. Each unsigned integer is assumed to be terminated by one or more spaces (blanks). There is no constraint on the particular fields in which the separate items are punched. This is called *free format input*.

We anticipate page 112 which explains the *A-FORMAT* in detail. In the example below the *FORMAT (80A1)* simply causes the character from each card column to be read and stored in the most significant end of the corresponding array element. In other words vector *L(80)* is made to contain an image of the punched card ~ one character per element ~ as sketched below. The sketch assumes a computer that can store up to two characters per integer location.

```
      INTEGER   L(80), SPACE
      DATA   SPACE / 37/
C
      READ (5, 100) L                 READ ALL 80 CHARACTERS
  100 FØRMAT  (80A1)
      I = 0
   10 I = I + 1                       END OF CARD
      IF (I .GT. 80) STØP
      K = INDEX (L(I))                INVOKES 'INDEX()'
      IF (K .EQ. SPACE) GØ TØ 10
      IF ((K .LT. 1) .ØR. (K .GT. 10)) STØP 1     NOT A DIGIT
      INTGER = 0                      INITIALIZE NEW INTEGER
      I = I - 1
   20 I = I + 1
      IF (I .GT. 80) GØ TØ 30
      K = INDEX (L(I))
      IF (K .EQ. SPACE) GØ TØ 30
      IF ((K .LT. 1) .ØR. (K .GT. 10)) STØP 2     NOT A DIGIT
      INTGER = 10 * INTGER + K - 1    ACCUMULATE AN INTEGER
      GØ TØ 20
   30 WRITE (6, 200) INTGER
  200 FØRMAT (1X, 10HINTEGER IS, I10)
      GØ TØ 10
      END
```



Some input:

```
123 0046 9999
```

the output:

```
INTEGER IS       123
INTEGER IS        46
INTEGER IS      9999
```

STATE TABLE

INPUT OF **ROMAN** *NUMERALS TO ILLUSTRATE USE OF THE* **DATA** *STATEMENT*

Assume all valid Roman numerals are composed of the following elements ≈ never more than one from each box. (In fact classical Rome preferred to see IIII rather than IV ≈ the subtractive principle was a later development.)

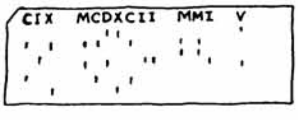
THOUSANDS	HUNDREDS		TENS		UNITS	
		D = 500		L = 50		V = 5
M = 1000	C = 100	DC = 600	X = 10	LX = 60	I = 1	VI = 6
MM = 2000	CC = 200	DCC = 700	XX = 20	LXX = 70	II = 2	VII = 7
MMM = 3000	CCC = 300	DCCC = 800	XXX = 30	LXXX = 80	III = 3	VIII = 8
ARBITRARY UPPER LIMIT MMM	CD = 400	CM = 900	XL = 40	XC = 90	IV = 4	IX = 9

The logic of this program is contained in the following *state table* (or *symbol-state table*):

"STATE"	"SYMBOL" M	D	C	L	X	V	I
→ 01	1000 & 02	500 & 03	100 & 09	50 & 05	10 & 10	5 & 07	1 & 11
02	1000 & 02	500 & 03	100 & 09	50 & 05	10 & 10	5 & 07	1 & 11
03	Error	Error	100 & 09	50 & 05	10 & 10	5 & 07	1 & 11
04	Error	Error	100 & 04	50 & 05	10 & 10	5 & 07	1 & 11
05	Error	Error	Error	50 & 06	10 & 10	5 & 07	1 & 11
06	Error	Error	Error	Error	10 & 06	5 & 07	1 & 11
07	Error	Error	Error	Error	Error	5 & 08	1 & 11
08	Error	Error	Error	Error	Error	Error	1 & 08
09	800 & 05	300 & 05	100 & 04	50 & 06	10 & 10	5 & 08	1 & 11
10	Error	Error	80 & 07	30 & 07	10 & 06	5 & 08	1 & 11
11	Error	Error	Error	Error	8 & 00	3 & 00	1 & 08

Take the Roman number **CIX** as an example: begin with a value of zero. You are initially in *state 1* (where the arrow is) so look down from *symbol* **C** and find *100 & 09* which says: "Add *100* to the value and change *state* to *09*". So add *100* to zero and move the arrow to *09*. Now look down from *symbol* **I** and find *1 & 11*. So add *1* to the value (*100 + 1 = 101*) and move the arrow to *state 11*. Finally look down from *symbol* **X** and find *8 & 00*. So add *8* to the value (*101 + 8 = 109*). The *00* means you've finished. Thus **CIX** is *109*.

The program is designed to read one card ≈ or row of data from a screen ≈ containing Roman numerals separated by spaces:

Some input: 

the output:

```
ROMAN NUMBER =      109
ROMAN NUMBER =     1492
ROMAN NUMBER =     2001
ROMAN NUMBER =        5
```

Here is the program:

```
      REAL      S(11,7)
      LOGICAL   SAME
      INTEGER   C(7), L(80), STATE, SYMBOL,SPACE, PRINTR, KARD
      DATA    SPACE, KARD, PRINTR / 1H  , 5, 6/
      DATA    C(1), C(2), C(3), C(4), C(5), C(6), C(7)
     1  / 1HM , 1HD , 1HC , 1HL , 1HX , 1HV , 1HI /
```

Elements of the state table are packed in *REAL* array *S(11,7)*. Each element is packed as *100.0 times the first part plus the second part*. A *REAL* array is used because on many computers the size of an integer array element is limited to 32767. Error states are represented by *−1.0* .

Here is the symbol-state table:

```
      DATA  S(1,1), S(1,2), S(1,3), S(1,4), S(1,5), S(1,6), S(1,7)
     1  / 100002., 50003., 10009., 5005., 1010., 507., 111. /
      DATA  S(2,1), S(2,2), S(2,3), S(2,4), S(2,5), S(2,6), S(2,7)
     1  / 100002., 50003., 10009., 5005., 1010., 507., 111. /
      DATA  S(3,1), S(3,2), S(3,3), S(3,4), S(3,5), S(3,6), S(3,7)
     1  /  -1. , -1. , 10009., 5005., 1010., 507., 111. /
      DATA  S(4,1), S(4,2), S(4,3), S(4,4), S(4,5), S(4,6), S(4,7)
     1  /  -1. , -1. , 10004., 5005., 1010., 507., 111. /
      DATA  S(5,1), S(5,2), S(5,3), S(5,4), S(5,5), S(5,6), S(5,7)
     1  /  -1. , -1. , -1. , 5006., 1010., 507., 111. /
      DATA  S(6,1), S(6,2), S(6,3), S(6,4), S(6,5), S(6,6), S(6,7)
     1  /  -1. , -1. , -1. , -1. , 1006., 507., 111. /
      DATA  S(7,1), S(7,2), S(7,3), S(7,4), S(7,5), S(7,6), S(7,7)
     1  /  -1. , -1. , -1. , -1. , -1. , 508., 111. /
      DATA  S(8,1), S(8,2), S(8,3), S(8,4), S(8,5), S(8,6), S(8,7)
     1  /  -1. , -1. , -1. , -1. , -1. , -1. , 108. /
      DATA  S(9,1), S(9,2), S(9,3), S(9,4), S(9,5), S(9,6), S(9,7)
     1  /  80005., 30005., 10004., 5006., 1010., 508., 111. /
      DATA  S(10,1), S(10,2), S(10,3), S(10,4), S(10,5), S(10,6), S(10,7)
     1  /  -1. , -1. , 8007., 3007., 1006., 508. , 111. /
      DATA  S(11,1), S(11,2), S(11,3), S(11,4), S(11,5), S(11,6), S(11,7)
     1  /  -1. , -1. , -1. , -1. , 800., 300., 108. /
```

The executable part, below, uses function *SAME(,)* from the previous example:

```
      READ (KARD, 100) L                      READ ALL 80 CHARACTERS AS IN PREVIOUS EXAMPLE
  100 FØRMAT (80A1)
      NEXT = 0
   10 NEXT = NEXT + 1                                            SKIP OVER SPACES
      IF (NEXT .GT. 80) STØP
      IF ( SAME ( L(NEXT), SPACE )) GØ TØ 10
C
      STATE = 1
      NUMBER = 0                              INITIALIZE
      NEXT = NEXT - 1
   40 NEXT = NEXT + 1
      IF((NEXT .GT. 80) .ØR. (SAME(L(NEXT),SPACE))) GØ TØ 50
      DØ 20  SYMBØL = 1, 7
      IF ( SAME ( L(NEXT), C(SYMBØL) ) ) GØ TØ 30
   20 CØNTINUE
      STØP 2                                  NOT M,D,C,L,X,V OR I
C
   30 ENTRY = S(STATE, SYMBØL)                PICK UP ENTRY FROM STATE TABLE
      IF (ENTRY .EQ. -1.0) STØP 3
      IF ((NEXT .GT. 3) .AND.
     1     SAME( SYMBØL, L(NEXT-1)) .AND.     CHECK NO MORE THAN
     2     SAME( SYMBØL, L(NEXT-2)) .AND.     3 SUCCESSIVE SYMBOLS
     3     SAME( SYMBØL, L(NEXT-3)) ) STØP 4  OF SAME KIND
C
      PART = AINT (ENTRY / 100.0)
      NUMBER = NUMBER + IFIX(PART)            ACCUMULATE
C
      STATE = IFIX ( ENTRY - 100.0*PART)      SECOND PART OF ENTRY
      IF (STATE .NE. 0) GØ TØ 40              ≡ NEW STATE
C
   50 WRITE (PRINTR, 200) NUMBER
  200 FØRMAT (1X, 14HRØMAN NUMBER =, I10 )
      GØ TØ 10
C
      END
```

EXERCISES

CHAPTER 9

9.1 Change the integer function *DAYS(MONTH, YEAR)* on page 86 so that it shares a named common block with a *BLOCK DATA* subprogram. Write this subprogram so that it initializes the variables *FEB* and *LEAP*, and the array elements *M(1)* to *M(12)*.

9.2 Recast the program on page 89 as a *subroutine* for reading integers written in free format. Include a logical argument *OK* to be returned *true* if the call succeeds; *false* otherwise. For example *OK* should be set *false* if the subroutine finds a letter among the digits (possibly letter O typed instead of a zero). If the subroutine reaches column 80 this should indicate the end of an integer: a new card should be read automatically if the subroutine is called again.

```
      CALL  IREAD ( N, ØK)
```

NEXT INTEGER DELIVERED IN VARIABLE 'N'

Allow for optional plus or minus signs if you want this subroutine to be really useful.

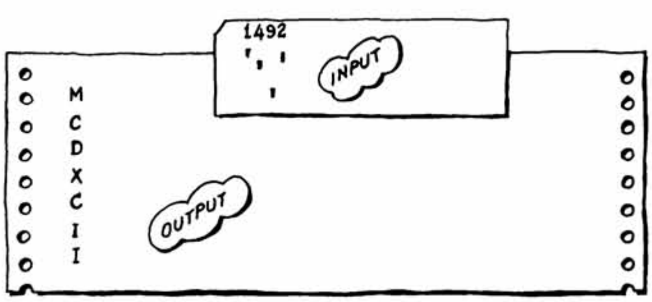
9.3 Write a converse program to the one on the previous page. The program should read unsigned decimal integers and print corresponding Roman numerals. Such a program doesn't require a symbol-state table and should be easier to devise than one to convert Roman numerals to Arabic. *DATA* statements may be useful; for example:

```
      DATA   K(1),  K(2),  K(3),  K(4),  K(5),  K(6),  K(7)
     1  /    1HM ,  1HD,   1HC ,  1HL ,  1HX ,  1HV,   1HI /
```

but remember your program would not be fully portable if you stored more than two characters in one integer variable:

```
      DATA   L(7) / 3HVII
```

Because we have not yet covered formatted output, print the results in a column down the left of the output page:




```
      WRITE  (6, 200)  K(I)
  200 FØRMAT (1X, A1 )
```

ONE CHARACTER

10

INPUT OUTPUT

READ

INPUT IN FORMATTED (READABLE) FORM
INPUT IN UNFORMATTED (BINARY) FORM

The introductory example showed:

```
      READ (5, 100) DIAM, HEIGHT, COVRG
  100 FORMAT   (3F10.0)
```

3 FIELDS; 10 COLUMNS EACH

which caused numbers on a waiting card (or card image) to be read into variables named *DIAM, HEIGHT, COVRG* respectively:

The general form of the *READ* statement is:

READ (*unit* , *format*) *list* — FORMATTED

or:

READ (*unit*) *list* — UNFORMATTED

where:

unit is either an integer constant (digits only) or the name of an integer variable (not an array element). *unit* denotes a connection to some peripheral device on which data are waiting to be read. The connection is made by command to the computer's operating system — perhaps by a *PROGRAM* command placed immediately before the main program. Any *unit* may be used for formatted *or* unformatted input; not a mixture of both. Conventionally *unit 5* denotes a card reader.

format may be omitted altogether — in which case the *READ* statement causes a complete *unformatted record* to be read. This record should be waiting on a peripheral device such as a magnetic-tape deck. The waiting unformatted record — possibly spread over several physical records such as blocks of magnetic tape — should match the *list* identically.

format, if present, may be the label (digits only) of a *FORMAT* statement (page 98) describing the layout of a *formatted record* or sequence of such records. The pattern of items in the waiting data should match not only the *list* identically but also the description given by the *FORMAT* statement.

format may also be the name of an integer array (conveniently an integer vector) containing the characters of the format description (page 102).

list (called the *input-output list* or *I/O list*) is a list of variables into which the waiting data are to be read. The example illustrates a simple *I/O* list: *DIAM,HEIGHT,COVRG* But there may also be array elements in the list, and their subscripts may be generated automatically. Page 96 defines the *I/O* list in detail.

Every *READ* statement causes a fresh record to be started; for example the next card to be read. Any items not demanded by the *I/O* list are **LOST** to the program. You can't retrieve subsequent items on the same card for example. A subsequent *READ* statement would cause the *next* card to go under the reading head.

WRITE

OUTPUT IN FORMATTED (READABLE) FORM
OUTPUT IN UNFORMATTED (BINARY) FORM

The introductory example showed:

```
      WRITE (6, 200) NPØTS
 200  FØRMAT (1X, 9H YØU NEED, I2, 5H PØTS )
```

which caused the value stored in *NPOTS* to be written as part of a single output record:

NPOTS [3] → *WRITE* → YOU NEED 3 POTS

FIELD OF TWO CHARACTER POSITIONS; I2

ONE LINE IS ONE 'PHYSICAL RECORD'

The general form of the *WRITE* statement is:

WRITE (*unit*, *format*) *list* ← *FORMATTED*

or:

WRITE (*unit*) *list* ← *UNFORMATTED*

where:

unit is as described opposite except that the peripheral device must be capable of output; for example a printer or tape deck but not a card reader. Conventionally *unit 6* denotes the line printer.

format may be omitted altogether – in which case the *WRITE* statement causes a complete *unformatted record* to be written on the nominated unit which may be connected to a magnetic-tape deck, disk file or paper-tape punch. The record to be written is as long as the *list* demands – possibly spread over several physical records (disk sectors or blocks of magnetic tape).

format, if present, may be the label (digits only) of a *FORMAT* statement (page 98) describing, in conjunction with the *I/O* list, one or more physical records (*e.g.* printed lines; punched cards).

format may also be the name of an integer vector which stores the format in character form (page 102).

list (*I/O* list) has the same form for output as for input. *list* is fully described on page 96.

GENERAL

CONCERNING READ *AND* WRITE *STATEMENTS:*
1. "READABLE" VERSUS "BINARY"   2. EMPTY I/O LISTS

A *formatted* record – on printer or screen – is in character form readable by eye. An *unformatted* record is more compact than the corresponding formatted record but would appear as gibberish if it were possible to send one to the printer. Unformatted records are used, typically, to transfer intermediate results between arrays and disk – *without* conversion to and from character form, hence without losing speed and accuracy during transfers. The information stays in binary.

In some circumstances the I/O list may be omitted from *READ* or *WRITE* statements:

```
      READ (5, 100)        <- SKIPS A COMPLETE PHYSICAL RECORD
 100  FØRMAT (3F 10.0)        (e.g. THE NEXT CARD) IGNORING THE
                              NOMINATED FORMAT
C
      READ (4)             <- SKIPS OVER THE BINARY RECORD
                              WAITING ON UNIT 4
      WRITE (7)               NOT ALLOWED WITHOUT AN I/O LIST
C
      WRITE (6,200)        <- WRITES HOLLERITH LITERALS IN THE FORMAT, BUT
                              TERMINATES ON MEETING A DESCRIPTOR (SUCH AS I4)
 200  FØRMAT (18H THIS GETS PRINTED, I4, 13H THIS DOESN'T )
```

I/O LIST

DENOTING A STREAM OF ITEMS TO BE READ OR WRITTEN (CONSTANTS & EXPRESSIONS ARE NOT ALLOWED)

The introductory example illustrated two simple *I/O lists* :

```
READ  (5, 100)  DIAM, HEIGHT, CØVRG
WRITE  (6, 200)   NPØTS                    <- LIST WITH ONE ITEM
```

The general form of *list* is as follows :

simple-list
(*simple-list*)
(*list* , *control* = *initial*, *terminal*)
(*list* , *control* = *initial*, *terminal*, *increment*)

or a *list* of such lists separated by commas , where:

simple-list consists of names of variables, names of arrays, names of array elements , or any combination of these. The names are separated by commas . Constants and expressions are not allowed in the I/O list.

control, *initial*, *terminal*, *increment*: these are as defined for the *DO loop* (page 40) and have the same implications as in a *DO loop*. The range of this *implied DO loop* is the preceding *list* contained within the same pair of brackets.

Here are more examples of *simple-lists* :

```
INTEGER     KARDS, LINPRN,  TAPE1,  TAPE2
DATA    KARDS, LINPRN,  TAPE1, TAPE2 / 5, 6, 4, 7/
```

```
READ ( KARDS, 300)  X, Y, A(2,1), A(2,2), A(I,J)
WRITE   (TAPE2)  MATRIX   <- COMPLETE ARRAY NAMED 'MATRIX' TRANSMITTED BY COLUMNS
```

The definition permits *list* to be a *simple-list* in brackets , but some Fortrans (including Fortran 77) object to unnecessary brackets:

```
READ (KARDS, 300) ( X, Y, A(2,1), A(2,2), A(I,J) )
WRITE (TAPE2)  ( MATRIX )
               ^ UNNECESSARY BRACKETS        77 BUG
```

Here is a statement illustrating the *implied DO loop*. This *must* be in brackets.

```
WRITE (LINPRN, 400)   ( VECTØR(I), I = 4, 10, 2 )
                      -> VECTOR(4), VECTOR(6), VECTOR(8), VECTOR(10)
```

And here is a *READ* statement illustrating a two-deep nest of implied *DO* loops and a *WRITE* statement illustrating a three-deep nest. The Fortran 66 standard does not forbid it, but some Fortrans do not allow a nest deeper than three.

```
READ  (TAPE1)  (( A(I,J) , I = 1, 60 ), J = 1, 100)               <- CAREFUL! LONG
WRITE (TAPE2)  ((( B(I, J, K), I = 1, 20), J = 1, 30), K= 1, 50)  <- RECORDS
```

Notice the recursive forms of the third and fourth definitions above ; *list* is defined in terms of *list*, thus permitting nested loops. These definitions account for the pattern of brackets and commas in the examples immediately above.

All four of the forms of the I/O list defined opposite have been illustrated individually, but notice that *list* may be a list of lists:

```
WRITE (LINPRN,500) (A(I), B(I), C(I), I = 1, 10), X, Y, Z
WRITE (LINPRN,600) (VEC(I), I = 1, 6), (COL(J), J = 1,3), P, Q
```

The items controlling the implied *DO* loop do not have to be integer constants; they may be names of integer variables. Page 50 describes the allowable forms of subscript involving integer variables.

```
WRITE (TAPE2)    (V(2*I + 3),  I = J, K, L )
```

(2*I +3) MAXIMUM COMPLEXITY OF A SUBSCRIPT OF V( )

VALUES CURRENT WHEN THIS WRITE STATEMENT IS OBEYED

It should be allowable to read subscripts and use them to address elements of an array on the assumption that Fortran works strictly from left to right along the I/O list:

```
READ (TAPE1)  I, J, A(I, J), B(J, I)
READ (TAPE2)  K, L, ( V(I), I = K, L )
```

but do not try to change parameters of an implied *DO* loop whilst it is still looping:

```
READ (TAPE1)  (K, L, V(I), I = K, L)
```

You are not allowed to transmit an array with *adjustable* dimensions using only its name (in the manner of array *MATRIX* illustrated opposite):

```
SUBROUTINE   COPY (A, I, J)
DIMENSION    A(I, J)
READ  (TAPE1)  A
WRITE  (TAPE2)  A
RETURN
END
```

but this subroutine could be corrected by changing the *READ* and *WRITE* instructions as follows:

```
READ (TAPE1) ((A (K, L), K = 1, I ), L = 1, J)
WRITE  (TAPE2)  ((A(K,L), K = 1, I ) , L = 1, J )
```

Many computers impose a limit on the length of an unformatted record, so find out what limit applies before attempting to transmit long unformatted records like those illustrated opposite. With formatted records it is a mistake to transmit more characters than the chosen medium can hold (e.g. 80 columns on a card; 132 characters on a printer). But the stream of items generated by an I/O list may be organized into a sequence of manageable records by *FORMAT* statements as later described.

FORMAT

DESCRIBING LAYOUT IN "READABLE" FORM: (EXHAUSTS THE I/O LIST ≈ BY "RESCAN" IF NECESSARY)

The introductory example showed:

```
    | READ (5, 100) DIAM, HEIGHT, CØVRG            (INPUT RECORD)
100 | FØRMAT  (3F10.0)

    | WRITE (6, 200) NPØTS                         (OUTPUT RECORD)
200 | FØRMAT (1X, 9H YØU NEED, I2, 5H PØTS )
```

where the lines labelled *100* and *200* illustrate *FORMAT* statements. These may be interspersed among the executable statements but *must always be labelled*.

Execution of a formatted *READ* or *WRITE* statement causes *a new record* to be started: the subsequent input or output in that record is "driven" by the nominated format. On input all items are in "readable" (character) form and have to be converted to binary form. On output all items are in binary form and have to be converted to character form. These conversions (from character to binary form and *vice versa*) are controlled by *descriptors* such as the *3F10.0* and *I2* in the examples above. Descriptors are dealt with exhaustively later in this chapter.

The form of the *FORMAT* statement is:

FORMAT (*description*)

where *description* may take any of the following forms:

- a list of *descriptors* separated by commas
- as above but with one or more commas replaced by a slash or group of slashes ≈ each slash starting a new record. (The slash or group of slashes may also be inserted at the end or the start of a list where there is no comma to replace, but such practice is not recommended.)
- as above but with one or more descriptors replaced by *descriptions* in brackets. The left bracket may be preceded by an integer saying how many times the content is implied (the *repeat count*). Omission of this integer implies once. (This definition is recursive; *description* is defined in terms of *description*.)

and where *descriptor* is any of the following (all of which are described in detail at the end of this chapter):

*n*P*t*F *w.d*	≈	fixed-point format
*n*P*t*E *w.d*	≈	*E*-format
*n*P*t*D *w.d*	≈	double-precision format
*n*P*t*G *w.d*	≈	generalized format
*t*I *w*	≈	integer format
*t*L *w*	≈	logical format
*t*A *w*	≈	character format
*w*H *hh...h*	≈	Hollerith literal
*w*X	≈	spaces (blanks)

and where:

*n*P is an optional scale factor, 10^n, described on page **110**. Omission implies $n = 0$; a scale factor of unity.

t is an optional integer saying how many *times* the descriptor is implied: omission implies once.

w is an integer specifying the number of character positions ~ the field width ~ of the item to be read or written. This integer must be greater than zero. Also *w* must be greater than *d*, where:

d is an integer specifying the number of digits assumed after the decimal point if no point is written in the data ~ or the number of decimal places to print on output. This integer must be included even if it is zero.

h each *h* (of which there are *w* in the field) is a Hollerith character ~ preferably from the Fortran 66 character set.

In the simplest formats there is a one-to-one correspondence between items in the I/O list and the various descriptors:

```
      READ (5,100) DIAM, HEIGHT, CØVRG
  100 FØRMAT     ( F10.0,   F10.0,   F10.0 )
```

but wherever a sequence of identical descriptors occurs the repeat count, *t*, may be used as illustrated by *3F10.0* opposite. Similarly the following two formats, *300* and *400*, are equivalent:

```
  300 FØRMAT  (2I4, 3F10.0, I4)
  400 FØRMAT  (I4, I4, F10.0, F10.0, F10.0, I4 )
```

and making use of the third form of *description* defined opposite, the following two formats are equivalent. The format labelled *500* is "nested":

```
  500 FØRMAT (2F10.2, 2(2F10.0, I4))
  600 FØRMAT (F10.2, F10.2, F10.0, F10.0, I4, F10.0, F10.0, I4)
```

In fact these two formats are not *quite* equivalent. If the I/O list supplied more than eight items there would be a difference in behaviour as explained below.

Descriptions may not be nested more than three deep:

```
  700 FØRMAT (1X,  3(F10.2, I4, 2(F10.0)))
```

DEPTH 1 DEPTH 2 DEPTH 3 NO DEEPER!

The I/O list should provide enough items to satisfy descriptors in the nominated format. Each record is closed as soon as the last element in the I/O list has found its descriptor. In this example the word *EXTRA* would not be added to the output record:

```
      WRITE  (6, 800) V1,V2
  800 FØRMAT (1X, 3F10.2, 6H EXTRA )
```

NOT ENOUGH VARIABLES TO SATISFY 3F10.2

But if the I/O list provides *more* items than the format can deal with then the format is automatically *rescanned* as depicted below:

```
      WRITE (6, 900) (VECTØR(I), I = 1, 30)
  900 FØRMAT (  1X,  F10.2  )
```

RESCAN FOR NEW RECORD

where each rescan initiates a *new record*. Thus on a line printer all thirty values above would appear as a column down the left-hand side of the output page. In a *nested* format the rescan (and start of new record) begins at the left bracket matching the *last but one* right bracket ~ taking account of the repeat count if there is one:

```
       WRITE (6, 1000) (VECTØR(I), I = 1, 30)
  1000 FØRMAT (1X, F10.0,  2(F10.1, F10.2),  F10.4 )
```

LAST BUT ONE RIGHT BRACKET

RESCAN FOR NEW RECORD

(continued)

FORMAT (CONTINUED) *BLANK RECORDS; MISMATCHING; CONTROLLING THE LINE PRINTER*

Every time format control meets a slash in place of a comma a new record is begun. A group of slashes thus produces *blank* records on output ~ or skips over complete records on input. Thus:

```
        DATA IVEC(1), IVEC(2),IVEC(3), IVEC(4), IVEC(5),IVEC(6)
     1   /      1 ,     2 ,     3 ,     4 ,     5 ,     6 /
        WRITE (6, 100) ( IVEC(J) , J = 1, 6 )
  100   FORMAT ( I4 , I4 / I4 , I4 , I4 // I4 / )
```

should produce:

```
O    1    2                FIRST RECORD                  O
O    3    4    5                         SECOND RECORD   O
O                          BLANK THIRD RECORD            O
O    6                                   FOURTH RECORD   O
O                          BLANK FIFTH RECORD ??         O
```

A group of k leading or trailing slashes should produce k blank records. A group of k slashes replacing a comma should produce $k-1$ blank records. Some Fortrans, however, treat leading and trailing slashes differently so it is best to avoid leading and trailing slashes for the sake of portability.

If an item of data being input fails to match its associated descriptor a warning message is usually printed. In some Fortrans this does not cause the run to stop but does leave the input variable undefined (containing rubbish).

```
        REAL  A
        READ  (5, 200) A        MISMATCH LEAVES 'A'
  200   FORMAT  (I4)            FULL OF RUBBISH
```

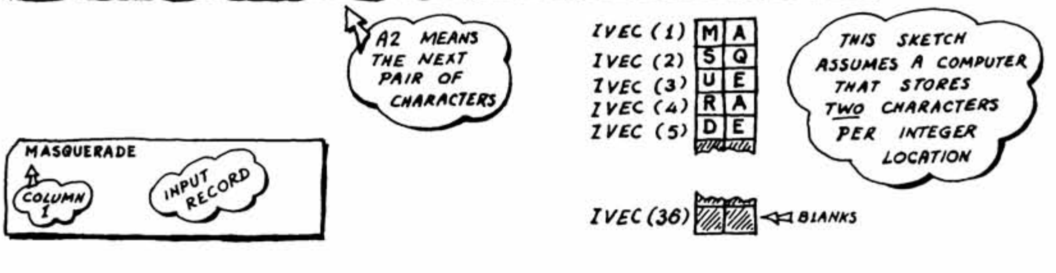
On output a mismatch usually causes the associated output field to be filled with asterisks or question marks:

```
        REAL  A
        A = 10.5
        WRITE (6, 300) A        MISMATCH
  300   FORMAT  (1X, I4, 9H MISMATCH)
```

```
O  ****  MISMATCH                                        O
```

However there is not necessarily a mismatch when a variable of one type is read or written using a format descriptor of another type. Both real and complex numbers may be transmitted using F, E and G descriptors. And because Fortran 66 has no variable of type CHARACTER (Fortran 77 does have this) Hollerith data have to be stored in the guise of integers:

```
        INTEGER  IVEC(36)
        READ  (5, 400)  IVEC
  400   FORMAT  (36A2)
```



A complex number requires *two* descriptors in the format:

```
      COMPLEX   C
      READ (5,500)  C
      WRITE(6,500)  C
  500 FORMAT (1X, F9.0, F10.0)
```

REAL PART IMAGINARY PART

```
127.5     3.2
COL 1     COL 10
```

```
127.5        3.2
```

If an output channel is connected to a line printer (conventionally unit 6 is so connected) *the first character is not printed*. The first character *position* is left blank; the character *itself* is used to control the line-feed mechanism of the printer as follows:

FIRST CHARACTER OF RECORD	CARRIAGE CONTROL
1 ← TYPICAL OF A FIRST RECORD	skip to the first line of a new page before printing the record
0 ← ZERO	skip *two* lines before printing the record
blank ← USUAL CASE	print this record on the next line
+ ← NOT EVERY FORTRAN OFFERS THIS FACILITY	print record on the *same* line (*i.e. overprint* previous record)
other characters	non-standard; but in some Fortrans treated as blanks

Previous examples of output formats begin with *1X* to ensure each record begins with a blank. Another convention is to replace the *1X* with *1H* .

```
  200 FORMAT ( 1X,
```

```
  200 FORMAT (1H ,
```

BLANK

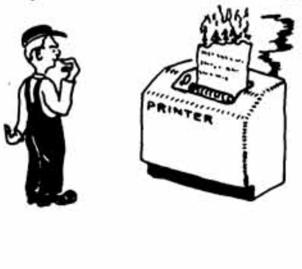
If a *FORMAT* statement consists only of *X* and *H* descriptors make sure the associated *WRITE* statement *has no I/O list*:

```
      DATA   MTAPE /4/
      WRITE (MTAPE, 100)  I
  100 FORMAT   ( 6H TRAMP, 2X )
```

Fortran would keep rescanning the format ⇔ hence keep writing "tramp, tramp, tramp" on the magnetic tape ⇔ vainly trying to find a descriptor for *I*. This could go on until the tape ran out. More destructive is:

```
      DATA   LPRINT /6/
      WRITE (LPRINT, 200)  I
  200 FORMAT   ( 132 (1H+) )
```

which could cut through the paper, mash the ribbon, even damage the printer. (I have not tested these routines and trust my readers won't.)



RUN-TIME FORMAT

ASSEMBLY VIA **DATA** *OR* **READ** *STATEMENTS*

As already explained; *format* in **READ**(*unit, format*) or **WRITE**(*unit, format*) may be the name of an integer array containing the format in character form. The keyword **FORMAT** is not itself included among the characters in this array.

Hollerith literals are not permitted in such formats because the number of trailing blanks in each array element depends on the make of computer being used. However it may be possible (but *not* standard) to get away with Hollerith literals one or two characters long.

The introductory example illustrated two formats:

```
  100  FØRMAT  (3F10.0)
  200  FØRMAT  (1X, 9H YØU NEED, I2, 5H PØTS )
```

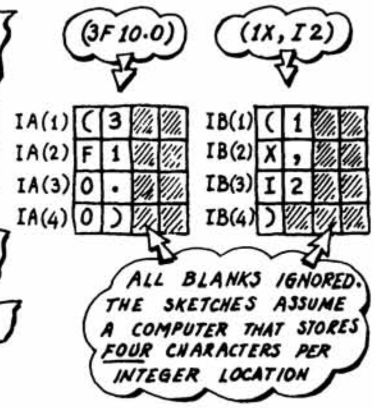
HOLLERITH LITERAL   HOLLERITH LITERAL

Apart from the two Hollerith literals these formats could have been assembled and used as follows:

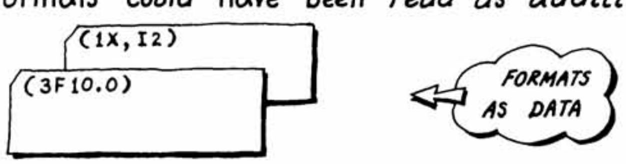
```
        INTEGER  IA(4), IB(4)
        DATA  IA(1), IA(2), IA(3), IA(4)
      1   /  2H(3, 2HF1, 2H0., 2H0) /
C
        DATA  IB(1), IB(2), IB(3), IB(4)
      1   /  2H(1, 2HX, , 2HI2, 1H)  /

        READ  (5, IA) DIAM, HEIGHT, CØVRG

        WRITE (6, IB) NPØTS
```



Or these formats could have been *read as additional data*:




```
        READ  (5, 300)  IA
        READ  (5, 300)  IB
  300   FØRMAT  ( A2 )
```

RESCAN FOR EACH PAIR OF CHARACTERS (THE 'A2' MEANS TWO HOLLERITH CHARACTERS)

The ingenious programmer can make the layout of results dependent on the magnitude of values to be printed. It is only necessary to manipulate the integers into which the Fortran 66 character set has been stored ≈ for example as on page 88. Remember you are *not* allowed to *assign* Hollerith constants:

```
        IA(3)  =  2H1.
```

but you can achieve the same effect by assigning integer values:

```
        DATA   IX / 2H1. /
        IA(3)  =   IX
```

GRAPH

AN EXAMPLE TO ILLUSTRATE A USE OF RUN-TIME FORMATS FOR PLOTTING

The following program is designed to plot a crude graph of $y = \cos(x)$ by printing asterisks as points on the curve. The scales have been chosen so that the graph would fit a modest 72-column printer (or VDU).

```
      INTEGER  RØW(61), STAR, BLANK, IPLACE, ANGLE, PRINTR
      REAL     RADIAN, CØSINE
C
      DATA  STAR, BLANK, PRINTR / 1H*, 1H , 6 /     BLANK
C
      WRITE (PRINTR, 100)                           OR WHATEVER UNIT IS USED FOR THE MODEST PRINTER
  100 FØRMAT (1H1, 41HGRAPH ØF CØSINE(X) FØR X=0 TØ 360 DEGREES)
C
      DØ 10 I = 1, 61
   10 RØW(I) = BLANK
C
      DØ 20 J = 1, 25                               1, 2, 3, ..., 25
      ANGLE = 15 * (J-1)                            0, 15, 30, ..., 360
      RADIAN = FLØAT(ANGLE) * 3.141593/180.0
      CØSINE = CØS(RADIAN)
      IPLACE = 30.0 * CØSINE  +  31.0               SCALE ROUGHLY TO CENTRE OF PAGE
      RØW(IPLACE) = STAR
      WRITE (PRINTR, 200)  ANGLE, RØW               ROW ≡ (ROW(I), I=1,61)
  200 FØRMAT (1H ,        I3   , 61A1 )
      RØW(IPLACE) = BLANK
   20 CØNTINUE
C
      STØP
      END
```

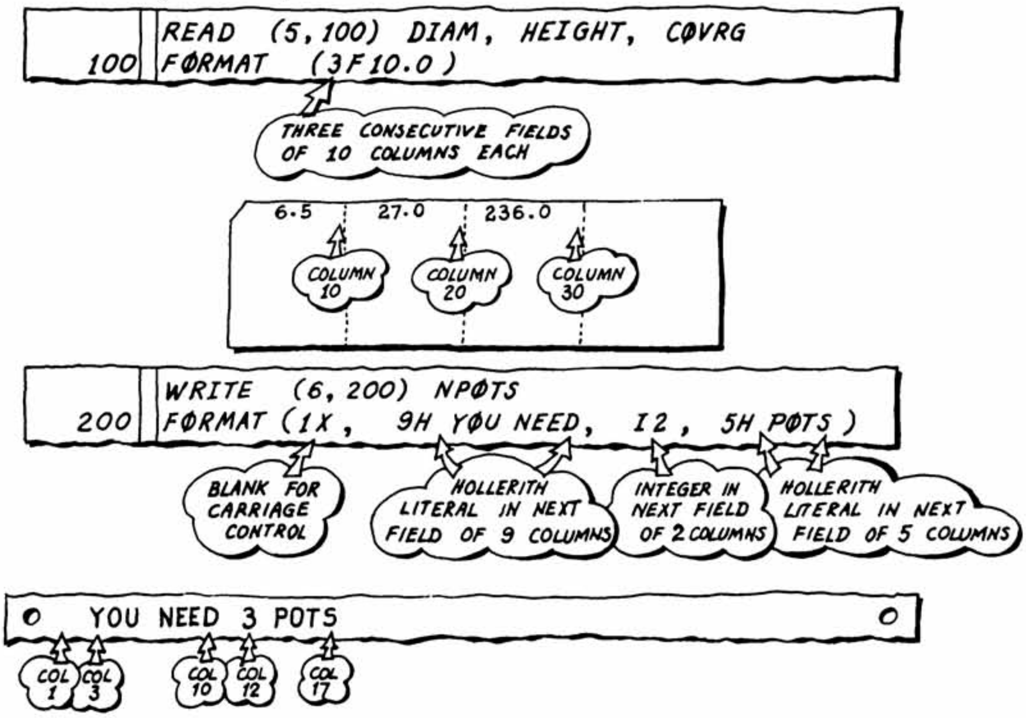
The output should look like this:

```
GRAPH OF COSINE(X) FOR X=0 TO 360 DEGREES
  0                                                            *
 15                                                           *
 30                                                        *
 45                                                    *        +1
 60                                              *
 75                                      *
 90                              *
105                       *
120                *             ZERO
135         *
150     *
165  *
180 *
195  *
210     *
225         *
240                *             ZERO
255                       *
270                              *
285                                      *
300                                              *
315                                                    *
330                                                        *
345
360                                                            *
```

DESCRIPTORS

RECAPITULATION ~ AND SUMMARY

The introductory example showed:



The programs (and fragments of programs) in earlier chapters used only:

- *1X* to make the printer start a new line
- *F10.0* for reading real numbers; *F10.2* for writing to two decimal places; *F10.4* to four places *etc.*
- *I2* for writing integers in a field of two columns; *I10* in a field of ten columns *etc.*
- Hollerith literals for annotating results
- *A2* for reading a pair of characters into a variable; *80A1* for reading a whole line of characters into successive elements of an integer array.

On the following pages all available descriptors (plus the scale factor *n*P) are explained in detail. The notation ~ already introduced on page 98 ~ is again used to summarize all descriptors as follows:

*n*P*t*F*w*.*d*	~	fixed-point format
*n*P*t*E*w*.*d*	~	*E*-format
*n*P*t*D*w*.*d*	~	double-precision format
*n*P*t*G*w*.*d*	~	generalized format
*t*I*w*	~	integer format
*t*L*w*	~	logical format
*t*A*w*	~	character format
*w*H*hh...h*	~	Hollerith literal
*w*X	~	spaces (blanks)

where: *nP* is an optional scale factor; *t* says how many times the descriptor is implied (omission implies once) ; *w* gives the field width; *d* gives the number of decimal places implied if no point is written ; *h* represents a Hollerith character ~ preferably from the Fortran 66 character set.

NUMBERS IN DATA

FORM OF NUMBERS CONVERTED BY F, E, D, G *DESCRIPTORS*

Data processed by *F, E, D* or *G* descriptors may be written in various ways. The essential thing is to *confine each number to the field specified by w* in its corresponding descriptor.

The simplest form allowed is the ordinary decimal number ≈ signed or unsigned:

```
 22.5       64.0       +22.5      -0.7
```

A trailing or leading decimal point is permissible but not nice:

```
 64.                  -.7
```

Numbers may also be written in exponent form:

`-1.5E6` or `-1.5E+6` for *-1500000*; `1.0E-2` for *0.001*

The *E* may be omitted if the exponent is signed:

```
 -1.5+6               1.0-2
```

Furthermore it is permissible to write *D* in place of *E* whether the number is destined to be a double-precision variable or not.

BLANKS in the field *count as zeros*. Thus:

```
 1 2 3 4 .    implies   010203040.00
```

where the leading zeros are ignored and the trailing zeros do no harm. A totally blank field is treated as *zero* by *F, E, D, G* and *I* descriptors alike.

It is safest **ALWAYS** to include a decimal point in numbers (especially those in exponent form) to be processed by *F, E, D, G* descriptors. The decimal point in the data *overrides an implied position* of a decimal point located by *d* (*Fw.d, Ew.d, etc.*). Not all Fortrans agree what *E7.3* implies if the corresponding item of data is, say :

```
   520E+6      or      520+6
```

CHECK THESE ON YOUR OWN MACHINE

but with *0.520E6* there is no doubt of the interpretation: 0.520×10^6.

FRUSTRATED OUTPUT

ITEMS TOO WIDE FOR THE FIELD

The value of *w* (in *Fw.d, Ew.d, ..., wX*) should specify a field wide enough to contain the number or message to be printed. If the field is too narrow, different Fortrans print different things ≈ usually causing information to be lost ≈ but few Fortrans cause the program to stop running. A typical outcome is a field full of asterisks or question marks: a frustrating result to receive:

```
      DATA VALUE / -12345.6 /
      WRITE (6, 100) VALUE
  100 FØRMAT (1X, 10H ANSWER IS, F6.1)
```

SHOULD BE AT LEAST F8.1

```
O ANSWER IS??????                                O
```

DESCRIPTOR F*w.d* — *FIXED-POINT FORMAT*

The item in the I/O list corresponding to this item should be of type *REAL*. The item may also be of type *COMPLEX* provided that it corresponds to a pair of such descriptors.

```
      REAL       R
      COMPLEX    C
      READ  (5, 100)  R, C
100   FORMAT  ( F10.0,  2F8.0 )
```

WRITING: The binary value in the computer is converted to character form and rounded to *d* decimal places. If the value is negative the number is prefixed with a minus sign. The number is output with as many preceding blanks as necessary to fit the number into a field of *w* character positions — right justified:

```
      REAL  A(4)
      DATA  A(1)  , A(2)  ,      A(3) ,      A(4)
     1  / -2368.4 , +12.0 , -17.90767, 37.5E-2 /
      WRITE  (6, 200)  (A(I), I = 1, 4)
200   FORMAT (1X,  F10.2 )
```

```
o     -2368.40                              o
o        12.00                              o
o       -17.91                              o
o         0.38                              o
o                                           o
```

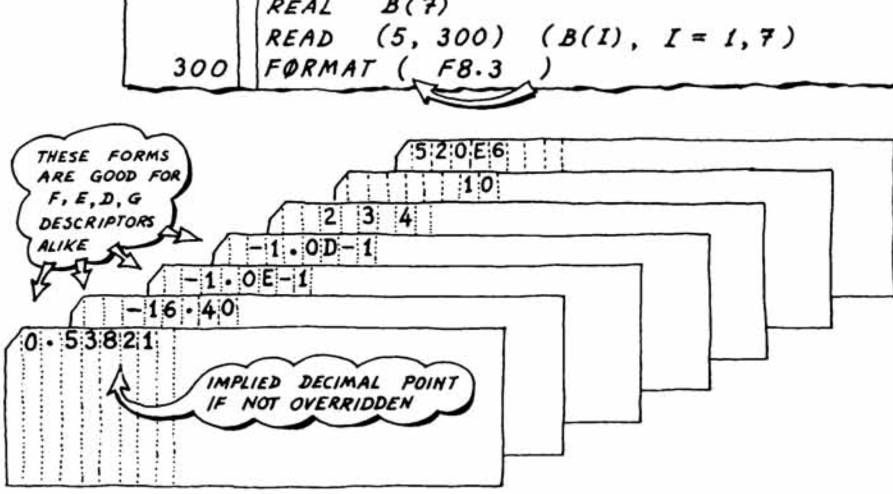
COL 1: CARRIAGE CONTROL

FIELD OF TEN

NOTE: IN FORMAT F6.0 THE VALUE -2368.4 WOULD BE PRINTED AS -2368. (DECIMAL POINT IN SIXTH COLUMN)

READING: The number in the next *w* character positions of the card (or card image) is converted to a *REAL* value and stored in the corresponding variable or array element nominated in the I/O list. If the number being read has no decimal point then the last *d* digits are assumed to represent the fractional part. In other words there is *an implied decimal point just before the last d digits*. If you punch a decimal point in the number, however, this *overrides* the implied decimal point. Always do this!

```
      REAL  B(7)
      READ  (5, 300)  (B(I), I = 1,7)
300   FORMAT (  F8.3  )
```



B(1)	0.53821
B(2)	-16.40
B(3)	-0.1
B(4)	-0.1
B(5)	203.04
B(6)	0.010
B(7)	?

DESCRIPTOR $E_{w.d}$

FLOATING-POINT FORMAT

The item in the I/O list corresponding to this descriptor should be of type *REAL*. The item may also be of type *COMPLEX* provided that it corresponds to a pair of such descriptors:

```
      REAL    R
      COMPLEX    C
      READ (5, 100)  R , C
  100 FORMAT  ( E10.0 , 2E8.0 )
```

WRITING: The binary value in the computer is converted to character form as follows:

. *digits* E + *ee*
. *digits* E − *ee*

where *digits* represents the *d* digits specified; *ee* represents a two-digit decimal exponent. Values are rounded to *d* decimal places and the resulting number is right justified in a field of *w* character positions.

Working from right to left the resulting number includes the two digits of the exponent, a sign for the exponent, the letter *E*, the *d* digits of the value, a decimal point, a minus sign or a blank in front of the value, enough blanks to complete the field width *w*. Some Fortrans replace the *E* with a *D*; others with a blank. Yet others replace the *E* with an extra digit to the exponent. Some Fortrans precede the decimal point with a zero; others with the first significant digit (e.g. *1.23E+00* instead of *.123E+01*). Some Fortrans precede positive values with plus signs instead of blanks. Such variations are permitted by the Fortran 66 standard. But if you make *w* greater than *d*+6 the field should be wide enough to contain the result whatever the style of output.

```
      REAL   A(4)
      DATA   A(1),        A(2),        A(3),     A(4)
     1   / 76.573, -58796.36, 37.5E-2, 0.0068 /
      WRITE (6, 200)  (A(I), I = 1, 4)
  200 FORMAT  (  1X,  E12.6  )
```

```
   .765730E+02
  -.587964E+05
   .375000E+00
   .680000E-02
```

COL 1: CARRIAGE CONTROL

FIELD OF TWELVE

READING: The *E* descriptor behaves as the *F* descriptor, but remember always to include a decimal point when writing an item of data. (One famous Fortran may multiply the number in the data by 10^d if the number before the *E* is integral !)

DESCRIPTOR D*w.d* — *FLOATING-POINT FORMAT FOR DOUBLE-PRECISION VALUES*

The item in the I/O list corresponding to this descriptor must be of type *DOUBLE PRECISION*. (Some Fortrans make *E* and *D* descriptors interchangeable but the standard is more restrictive.)

```
      DØUBLE PRECISIØN   DBL
      READ  (5, 100)    DBL
  100 FØRMAT   ( D12.0 )
```

WRITING: The binary value in the computer is converted to character form as follows:

> .*digits* D+ *ee*
> .*digits* D− *ee*

where *digits* represents the *d* digits specified; *ee* represents a two-digit exponent. Values are rounded to *d* decimal places and the resulting number is right justified in a field of *w* character positions.

The variations in style of output described on the previous page (in particular the replacement of *E* by *D* and vice versa) may apply to the *D* descriptor as well. These variations are all admissible in the Fortran 66 standard. Again, if you make *w* greater than *d*+6 the field should be wide enough to contain the result whatever the style of output.

```
      DØUBLE  PRECISIØN  DP(4)
      DATA        DP(1),          DP(2),    DP(3),  DP(4)
     1  / 76.573D0 , -587963.66D0, 37.5D-2, 68D-4 /
      WRITE  (6, 200)  (DP(I), I = 1, 4)
  200 FØRMAT (  1X,   D12.6  )
```

```
  .765730D+02
-.587964D+05
  .375000D+00
  .680000D-02
```

COL 1: CARRIAGE CONTROL

FIELD OF TWELVE

READING: The *D* descriptor behaves as the *F* and *E* descriptors, but remember the corresponding item in the I/O list must be of type *DOUBLE PRECISION*.

DESCRIPTOR $G_{w.d}$ GENERALIZED FORMAT

The item in the I/O list corresponding to a *G* descriptor should be of type *REAL*. The item may also be of type *COMPLEX* provided that it corresponds to a pair of such descriptors.

```
          REAL     R
          COMPLEX     C
          READ  (5, 100)  R, C
100       FORMAT  ( G10.0,  2G8.0)
```

WRITING: The binary value in the computer is converted to one of two character forms:

- if the absolute value lies between 0.1 and 10^d then in fixed-point form with four blanks to the right (reducing the effective width of field to $w-4$ character positions). The best use of the remaining field is made as illustrated below
- otherwise in *E*-format right justified in the full field width of *w* character positions.

```
          REAL  A(7)
          DATA    A(1),      A(2),     A(3),     A(4)
     1       / 0.0666,   0.666,   6.66,   66.6  /
          DATA    A(5),      A(6),       A(7)
     1       /  666.0,    6660.0,    66600.0  /
          WRITE  (6, 200)  (A(I), I= 1,7 )
200       FORMAT (  1X,   G12.4  )
```

```
      .6660E-01
     0.6660
      6.660
      66.60
      666.0
      6660.
      .6660E+05
```

COL 1: CARRIAGE CONTROL

FIELD OF TWELVE

READING: The *G* descriptor behaves as the *F*, *E* and *D* descriptors.

SCALE FACTOR nP

IN CONJUNCTION WITH F, E, D, G DESCRIPTORS (DANGEROUS !)

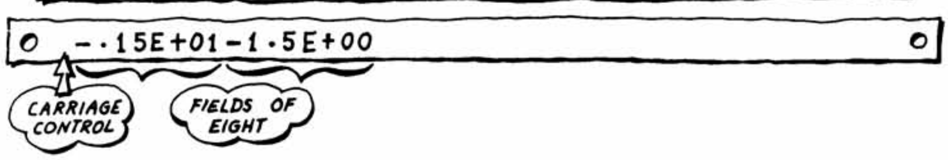
The n is an integer constant which may be negative, zero or positive. The scale factor is 10^n.

WRITING: In *F*-format the number printed is 10^n times the value stored:

```
      DATA  B / 1.5 /
      WRITE (6, 100) B, B
  100 FØRMAT ( 1X,  3PF8.2 ,  -2PF8.2 )
```

In *E*-format and *D*-format the decimal point is shifted n places to the right (when n is positive) and the exponent is reduced by n. In other words the number printed is *not scaled* at all but simply altered in appearance:

```
      DATA  Q / -1.5 /
      WRITE (6, 200) Q, Q
  200 FØRMAT ( 1X,  E8.2 ,  1PE8.2 )
```



In *G*-format the scale factor has no effect on numbers printed in the style of *F*-format (range 0.1 to 10^d). For numbers outside this range the effect of the scale factor is the same as that just described for the *E*-format.

READING: When you write an item of data with no exponent then the value of that item is divided by 10^n. When you write an item of data *with* an exponent then the scale factor has no effect on the value stored:

```
      READ (5, 300) (A(I), I = 1, 3)
  300 FØRMAT ( 2PF6.2,  F6.2 ,  F6.2 )
```

The value of n is assumed to be zero when not specified at all ($10^0 = 1$) but once a value for n is specified it *remains in force* for all other descriptors in the same *FORMAT* statement (even for descriptors in deeper nested brackets and in all rescans) until *re-set* by another *nP*.

ZERO NULLIFIES THE SCALE FACTOR

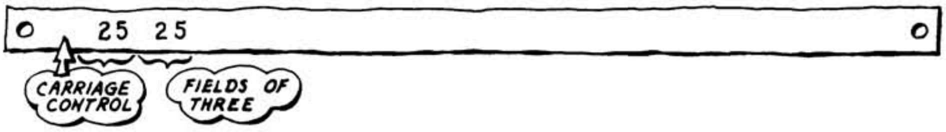
```
  400 FØRMAT ( 2PF6.2,  0PF6.2 )
```

DESCRIPTOR Iw

ONLY INTEGERS MAY BE TRANSMITTED USING THIS DESCRIPTOR

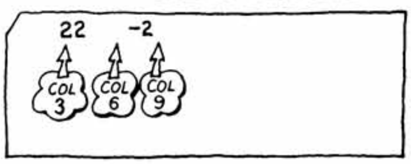
WRITING: Integer values to be printed are right justified in a field of *w* character positions:

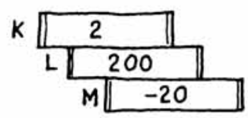
```
      INTEGER    K
      DATA       K / 25 /
      WRITE  (6, 100)  K , K
100   FØRMAT   ( 1X,  2I3 )
```



READING: Items are assumed to be right justified within a field width of *w* ⇒ blanks being treated as zeros. Negative values should be preceded by a minus sign. Plus signs in front of positive values are allowable but not necessary.

```
      INTEGER   K, L, M
      READ  (5, 200)  K,  L,  M
200   FØRMAT   ( 3F3)
```



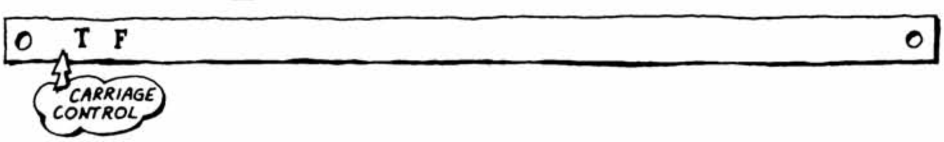


DESCRIPTOR Lw

ONLY LOGICAL VALUES MAY BE TRANSMITTED WITH THIS DESCRIPTOR

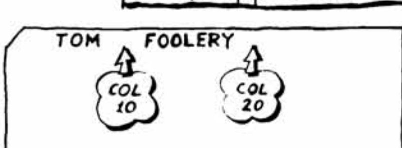
WRITING: In a field of *w* character positions the letter *T* or *F* is right justified. *T* signifies the Boolean value *true* and *F* signifies *false*.

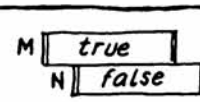
```
      LØGICAL     M , N
      DATA       M, N / .TRUE. , .FALSE. /
      WRITE  (6, 300)  M , N
300   FØRMAT  (1X,  L1,  L2 )
```



READING: Inside a field of *w* character positions the first encounter of the letter *T* or *F* signifies the Boolean value *true* or *false* respectively. The Boolean value is assigned to the corresponding logical variable nominated in the I/O list:

```
      LØGICAL    M , N
      READ  (5, 400 )  M , N
400   FØRMAT   (2L10)
```





DESCRIPTOR A_w

FOR DEALING WITH CHARACTERS

Unlike Fortran 77, Fortran 66 has no variable of type CHARACTER. Instead characters may be stored in variables of any type ≈ and it depends on the computer how many characters each type of variable can hold. On a typical 16-bit computer an integer variable may hold two characters, a real variable four, a double-precision or complex variable eight. For the sake of generality we speak of a variable (hence also an array element) holding g characters in the illustrations below. In the examples g is assumed to be 4, and a blank (or space) is depicted as ▨.

WRITING: If the field width w is greater than or equal to g then g characters are right justified in the field and enough blanks added to fill the field:

```
      DATA  V(1), V(2), V(3) , V(4), V(5)
     1  / 4HFØUR, 1H  , 4HCHAR, 4HACTE, 2HRS /
      WRITE (6,100) (V(I), I = 1,5)
  200 FØRMAT (1X, I6 , 4I4 )
```

$g=4$

V(1)	F	O	U	R
V(2)	▨	▨	▨	▨
V(3)	C	H	A	R
V(4)	A	C	T	E
V(5)	R	S	▨	▨

$w \geq g$

```
0    FOUR    CHARACTERS
```


If the field width is less than g then the left-most characters are sent to fill the field, those to the right being lost:

```
      WRITE (6,200) (V(I), I = 1,5)
  200 FØRMAT (1X , 5I2 )
```

$g=4$

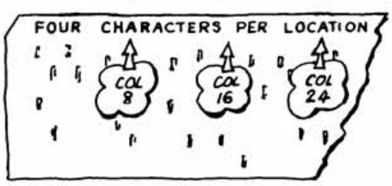
V(1)	F	O	U	R
V(2)	▨	▨	▨	▨
V(3)	C	H	A	R
V(4)	A	C	T	E
V(5)	R	S	▨	▨

$w < g$

```
0 FO  CHACRS
```


READING: If the field width w is greater than g then only the right-most g characters are stored in the associated variable, the foremost $w-g$ characters being skipped over:

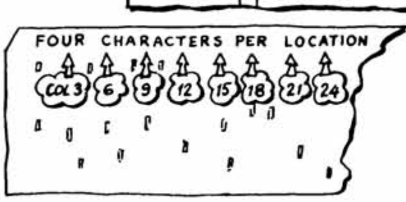
```
      READ (5, 300)  (V(I), I = 1,3)
  300 FØRMAT    (3I8)
```



V(1)	▨	C	H	A
V(2)	E	R	S	▨
V(3)	L	O	C	A

If the field width is less than or equal to g then all w characters are left justified in the variable, the remainder of the variable being filled with blanks.

```
      READ  (5, 400)  (V(I), I = 1, 8)
  400 FØRMAT    (8I3 )
```



V(1)	F	O	U	▨
V(2)	R	▨	C	▨
V(3)	H	A	R	▨
V(4)	A	C	T	▨

w

V(5)	E	R	S	▨
V(6)	▨	P	E	▨
V(7)	R	▨	L	▨
V(8)	O	C	A	▨

HOLLERITH LITERAL wH$hh...h$ FOR CAPTIONS

WRITING: The w characters to the right of letter H define a field of width w and are sent to fill that field precisely. Blanks within this field are significant characters. The set of characters that may be used in a Hollerith literal is more extensive than that defined as the Fortran 66 character set and more extensive than the Fortran 77 set (page 20) but for portable programs it is best not to use non-standard characters.

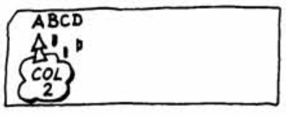
```
      WRITE  (6, 100)
  100 FØRMAT(1X, 26H THIS IS A HØLLERITH LITERAL)
```

```
O  THIS IS A HOLLERITH LITERAL                       O
```

CARRIAGE CONTROL

READING: According to the Fortran 66 standard the w characters of the Hollerith field on the input medium should *replace* the corresponding w characters appearing in the *FORMAT* statement as illustrated here:

```
      READ  (5, 200)
  200 FØRMAT (1X,  4H**** )
      WRITE  (6, 200)
```




```
o  ABCD                o
```

NOT ****

Although this is standard Fortran 66 the practice is not recommended to those wanting to write fully portable programs. There are some (otherwise standard) Fortrans that do not offer this facility of replacement. Try it on your own machine.

BLANKS (SPACES) wX FOR PRINTING SPACES OR SKIPPING DATA

WRITING: A total of w blanks are sent to the output record on each wX encountered. See page 101 about carriage control.

```
      WRITE  (6, 300)
  300 FØRMAT (1X,3HHØW, 1X,3HNØW, 1X, 5HBRØWN, 3X, 3HCØW)
```

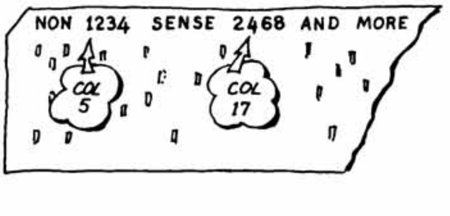
```
O  HOW NOW BROWN   COW                               O
```

CARRIAGE CONTROL

3 BLANKS

READING: A total of w characters (including blanks) on the input record are skipped over on each wX encountered:

```
      READ  (5, 400)  J, K
  400 FØRMAT  (4X,  I4,  8X,  I3 )
```



J 1234

K 468

THE WORDS ON THE CARD ARE SKIPPED OVER

FREE FORMAT

AN EXAMPLE TO ILLUSTRATE THE AVOIDANCE OF TROUBLESOME DESCRIPTORS

Input in Fortran is based on the concept of the punched card but nowadays very many (if not most) computer installations offering Fortran do not use punched-card equipment. It can be awkward trying to ensure that numbers typed at a VDU ≈ or on punched tape ≈ appear in the correct fields. The solution to this problem adopted at many installations is to provide a special format descriptor or special *READ* statement which simply reads *the next number* whatever field it occupies. The disadvantage of this solution is that it brings non-standard features into Fortran ≈ and these non-standard features vary in specification from one computer to another.

The subroutine described here illustrates a way of reading numbers in free format. The subroutine is nevertheless written in *standard* Fortran 66 and should be fully portable.

On each call the subroutine reads the next waiting number as a *REAL*. The waiting number may be prefixed with a plus or minus sign and may contain a decimal point. This subroutine does not allow numbers written in exponent form, nor does it permit a leading or trailing decimal point:

−1.27E3 *−.5* *38.*

Numbers should be written separated by one or more spaces as follows:

−1270 *−0.5* *38.0* (or just *38*)

The subroutine (which must be initialized: see note opposite) may be called as illustrated below:

```
      LØGICAL   ØK
      REAL      V
      CALL  INPUT(V, ØK)
```

if *OK* returns *true* the item that was waiting is returned by *V*. If *OK* returns *false* it means the waiting item was wrongly formed. However, the pointer is then restored to where it was before the call ≈ thus making it possible to attack the item with another subroutine (for instance a routine for reading keywords).

The subroutine *INPUT* seeks *the next item*. This subroutine reads a *new* card (or card image) if there are no more items in the current image.

The logic of this subroutine is based on the following state table:

INITIAL STATE

STATE	SYMBOL					
	1: DIGIT	2: +	3: −	4: •	5: SPACE	6: END OF RECORD
1	ACTION 1: ACCUMULATE DIGIT. NEXT STATE=3	NEXT STATE=2	ACTION 3: NEGATE THE RESULT. NEXT STATE=2	EXIT FALSE	NEXT STATE=1	ACTION 4: READ A NEW CARD. NEXT STATE=1
2	ACTION 1: ACCUMULATE DIGIT. NEXT STATE=3	EXIT FALSE	EXIT FALSE	EXIT FALSE	EXIT FALSE	EXIT FALSE
3	ACTION 1: ACCUMULATE DIGIT. NEXT STATE=3	EXIT FALSE	EXIT FALSE	NEXT STATE=4	EXIT TRUE	EXIT TRUE
4	ACTION 2: ACCUMULATE FRACTION. NEXT STATE=5	EXIT FALSE	EXIT FALSE	EXIT FALSE	EXIT FALSE	EXIT FALSE
5	ACTION 2: ACCUMULATE FRACTION. NEXT STATE=5	EXIT FALSE	EXIT FALSE	EXIT FALSE	EXIT TRUE	EXIT TRUE

```
      SUBRØUTINE  INPUT (VALUE, STATUS)
      LØGICAL   STATUS, SAME
      INTEGER   RØW(80), L(14), T(5,6), READER, SYMBØL, STATE
     1          ENTRY, K, ACTIØN, FIRST, NEXT
      REAL      VALUE, DIVISR, SIGN, DIGIT                         DIGITS
      CØMMØN / PØINT / NEXT
      DATA      READER/5/
      DATA  L(1), L(2), L(3), L(4), L(5), L(6), L(7), L(8), L(9), L(10)
     1  /   1H0 , 1H1 , 1H2, 1H3, 1H4, 1H5, 1H6, 1H7, 1H8, 1H9 /
      DATA  L(11), L(12), L(13), L(14)                 SYMBOLS
     1  /   1H+ , 1H- , 1H. , 1H    /
      DATA  T(1,1) , T(1,2), T(1,3), T(1,4), T(1,5), T(1,6)
     1  /     13 ,    72,     32 ,    60 ,    71 ,    41 /
      DATA  T(2,1) , T(2,2), T(2,3), T(2,4), T(2,5), T(2,6)
     1  /     13 ,    60,     60,     60,     60 ,    60 /
      DATA  T(3,1) , T(3,2), T(3,3), T(3,4), T(3,5), T(3,6)
     1  /     13 ,    60,     60,     74 ,    50 ,    50 /
      DATA  T(4,1) , T(4,2), T(4,3), T(4,4), T(4,5), T(4,6)
     1  /     25 ,    60,     60,     60 ,    60 ,    60 /
      DATA  T(5,1) , T(5,2), T(5,3), T(5,4), T(5,5), T(5,6)
     1  /     25 ,    60,     60,     60 ,    50 ,    50 /
C                                                          SYMBOL
      VALUE  =  0.0            FIRST EXECUTABLE            STATE
      DIVISR  =   1.0             STATEMENT                TABLE
      SIGN   =  +1.0
      STATUS  =   .FALSE.      'NEXT' MUST BE INITIALIZED BEFORE
      FIRST  =   NEXT               FIRST CALL TO 'INPUT'
      STATE  =   1              e.g.   BLOCK DATA
C                                      CØMMØN / PØINT / NEXT
                                       DATA   NEXT / 81/
    7 NEXT  =  NEXT + 1                END
      SYMBØL  =  6
      IF ( NEXT  .GT.  80)  GØ TØ  30
      K = RØW (NEXT)
      DØ  10  I = 1, 14
      IF  ( SAME(K , L(I)))  GØ  TØ  20
   10 CØNTINUE
      GØ  TØ   6          CHARACTER NOT ALLOWED
C
   20 SYMBØL  =  1
      DIGIT  =  I - 1
      IF ( I .GT. 10)  SYMBØL = I - 9
   30 ENTRY  =   T ( STATE ,  SYMBØL)
      ACTIØN   =   ENTRY / 10
      STATE    =   ENTRY  -  10 * ACTIØN
      GØ TØ   (1, 2, 3, 4, 5, 6, 7) ,  ACTIØN
C
    1 VALUE  =  10.0 * VALUE  +  DIGIT
      GØ  TØ   7
C
    2 VALUE  =  10.0 * VALUE   +  DIGIT
      DIVISR  =   10.0 * DIVISR
      GØ  TØ   7
C
    3 SIGN  =  -1.0
      GØ  TØ   7
C
    4 READ (READER,  100)   RØW
  100 FØRMAT    (80A1)
      NEXT = 0
      FIRST   =  0
      GØ  TØ   7
C
    5 VALUE  =  SIGN * VALUE / DIVISR
      STATUS  =  .TRUE.
      RETURN                  TRUE EXIT
C
    6 NEXT = FIRST            FALSE EXIT
      RETURN                  (POINTER RESTORED)
C
      END
```

EXERCISES

CHAPTER 10

10.1 Work through some of your earlier programs replacing the rudimentary output with neatly formatted results. In particular let input values feature among the results. For example the results from Exercise 9.3 should appear something like this:

```
o  ENCODING  ROMAN  NUMERALS                       o
o                                                  o
o      1492  =  MCDXCII                            o
o                                                  o
o         5  =  V                                  o
```

10.2 Write a program to plot several functions simultaneously. For example $y = sin(x)$, $y = cos(x)$ and $y = tan(x)$ using a different symbol for points on each curve : for example * and + and •

10.3 Extend subroutine *INPUT* (page 115) to cope with numbers written in exponent form. This involves an extension to the state table.

10.4 Write a subroutine to read a "keyword" and match it against a list of allowable keywords stored in the program. For example let the list be *RECTANGLE*, *TRIANGLE*, *CIRCLE* and let the first *two* characters suffice for a match : *RE*, *TR*, *CI*. If the subroutine discovers the word *RECTANGLE* (or *RECT* or *RE* or even *REK*) it should return *1*. If the routine finds the word *TRIANG* it should return 2, and so on. If the routine fails to find a match in the list for the item just read then the routine should return zero (or set a logical argument *false*) and set its pointer back where it was when the routine was called (*i.e.* follow the principle used in subroutine *INPUT*).

Include this subroutine ≈ together with subroutine *INPUT* ≈ in the example program on page 45. The data for this program could be recast as follows:

```
TRIANGLE    15.4   16.8   21.95
RECTANGLE     13.67    10
TR   3  4  5
CIRC    15.9    <=== ASSUMING "NO" IS ADDED
NØMØRE               TO THE LIST OF KEYWORDS
```

11

FILES

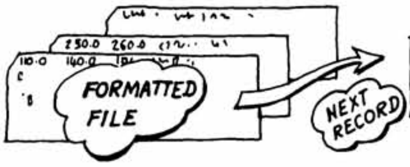
FORMATTED FILES
UNFORMATTED FILES
ENDFILE, REWIND, BACKSPACE
EXERCISES

FORMATTED FILES

SOME CONCEPTS AND TERMINOLOGY

A formatted file consists of *physical* records: each I/O list specifies a *logical* record.

A deck of cards punched in readable (character) form is a formatted file:




```
    READ (KARDS, 100) (V(I),I=1,48)
100 FØRMAT (  8F10.0  )
```

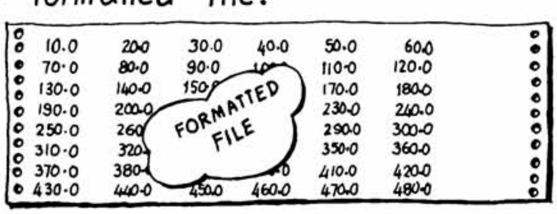
EACH RESCAN BEGINS THE NEXT RECORD

The length of each record is limited by the *external medium* (in this case the punched card). It would be a mistake to write the format as (48F10.0) because one eighty-column card cannot contain 48 fields each of 10 columns. The FORMAT statement is responsible for chopping up the I/O list – *i.e.* the logical record – into manageable physical records by rescans or slashes or both.

A set of printed output is also a formatted file:

```
    WRITE (LINPRN,200) (V(I),I=1,48)
200 FØRMAT ( 1X, 6F10.1 )
```

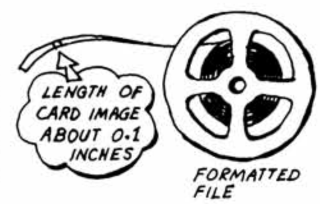
EACH RESCAN BEGINS THE NEXT RECORD



The length of each record is again limited by the external medium (in this case the longest line the particular printer can print). It would be a mistake to write the format as (1X,8F10.1) if the printer had a line length of 72 columns. Again the FORMAT statement is responsible for chopping the I/O list into manageable physical records by rescans or slashes or both.

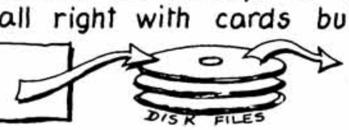
A formatted file may also reside on punched paper tape or magnetic tape or on a disk file. In such cases the I/O list in conjunction with the FORMAT statement *again* determines the length of record – but the external medium does not impose any practical constraint on length. (The computer does divide magnetic tape into "blocks" but this should not have to concern the Fortran programmer directly.) In the following example the record length is chosen as eighty columns – in other words as a "card image".


```
    INTEGER LINE(80)
    DØ 10 I=1,6
    READ (KARDS, 300) LINE
10  WRITE(ITAPE, 300) LINE
300 FØRMAT (80A1)
```



Beware of writing records in one format and later trying to read them back in another. This is because many operating systems do not "pad out" short records with blanks to represent card images. Instead they keep account of the number of characters actually in the record. Below is shown an attempt to read twelve characters where only ten were written: all right with cards but not with disk files.

```
    WRITE(IFILE, 400) X
400 FØRMAT ( F10.2)
```



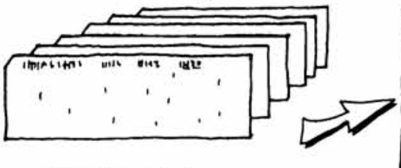

```
    READ(IFILE, 500) Y
500 FØRMAT ( F12.0)
```

UNFORMATTED FILES

MORE CONCEPTS AND TERMINOLOGY

An *unformatted file* consists of *logical* records, each as long as the I/O list demands. (Some operating systems do impose a limit on length but it is usually quite big.) An unformatted file cannot be sent to a line printer; if it *could* the output would be unreadable because unformatted files are coded in binary form.

A formatted file may be converted to an unformatted file on a suitable medium (say magnetic tape) by means of a program:




```
      DIMENSIØN  LINE(80)
      DØ 10  I = 1, 6
      READ  (KARDS, 100) LINE
   10 WRITE (ITAPE) LINE
  100 FØRMAT  ( 80A1 )
```

In the above example each unformatted record is the binary equivalent of eighty integers ~ a "binary card image" in which each integer stores one Hollerith character in its most significant bits.

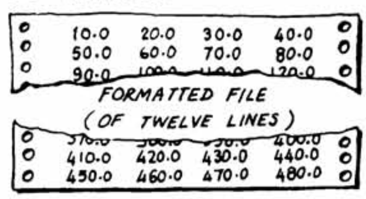
Conversely an unformatted file may be converted to a formatted file by a program. To write this you have to know:

- the number of records in the unformatted file
- the number of items in each logical record (they could all be of different length !)
- the *type* of each such item (these could all be different too)
- the limiting length of formatted record to be written

Assuming the file of six logical records illustrated above ~ and assuming the data listed opposite ~ the program could be as follows:


```
      DIMENSIØN  LINE ( 80)
      DØ 20  I = 1, 6
      READ  (ITAPE ) LINE
   20 WRITE (LINPRN , 200) LINE
  200 FØRMAT ( 1X, 40A1 )
```

TWO "HALF CARDS" PER CARD IMAGE



Finally we illustrate a straightforward application of unformatted files used as "backing store" in the "number crunching" type of problem. *A, B, C* may be big multi-dimensional arrays accounting for very long records.

```
      READ  (IFILE) A
      READ  (JFILE) B
      CALL  CRUNCH( A, B, C)
      WRITE  (KFILE) C
```

BUT you may do more than just read and write files serially; Fortran 66 provides ENDFILE, REWIND and BACKSPACE (see overleaf). These offer only limited control of files but the deficiencies in Fortran 66 are overcome in Fortran 77. Here you may detect an end-of-file; make access to any record of a file directly; inquire into the status of a file (*e.g.* whether it exists) and take action if there is a misread. Consult your own Fortran user's manual and check if any such facilities offered conform to Fortran 77 standards.

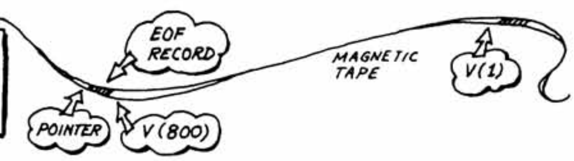
ENDFILE, REWIND, BACKSPACE

AUXILIARY I/O STATEMENTS

As their individual names suggest, the auxiliary I/O statements were originally designed for manipulating magnetic tapes. They are equally applicable to manipulating *files* stored on a disk or other modern secondary-storage device. But the auxiliary I/O statements would make no sense in controlling the more traditional devices: you can't "rewind" or "backspace" a card punch, card reader, paper-tape punch or reader, or line printer.

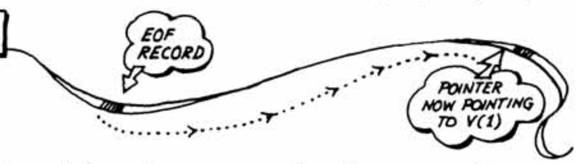
ENDFILE puts a dab of conceptual paint on a magnetic tape. This is called an *end-of-file record* (EOF). In bygone days the EOF was useful when the reel of tape was taken off the computer's tape deck and mounted on a special deck for driving a line printer ("off-line printing" in the jargon). The EOF — the dab of conceptual paint — was recognized by the printing installation which would throw a few blank pages before printing the next file. But Fortran 66 *does not recognize an EOF record when reading a file*. If an EOF is encountered during input different Fortrans do different things: some give up altogether.

```
      DATA    MYFILE /7/
      WRITE (MYFILE) (V(I), I= 1,800)
      ENDFILE   MYFILE
```



REWIND positions a conceptual pointer at the first record of the nominated file. If this file happens to be already rewound then *REWIND* acts as a "do nothing" statement. (There is danger using *REWIND* on magnetic tapes: some Fortrans rewind the entire tape rather than just one file. With disk files, however, the *REWIND* statement should behave properly.)

```
      REWIND   MYFILE
```



BACKSPACE causes the conceptual pointer to move back over the previous record (EOF record no exception) of the nominated file. If this file happens to be rewound (the pointer at the first record) then *BACKSPACE* acts as a "do nothing" statement.

```
      BACKSPACE   MYFILE
```

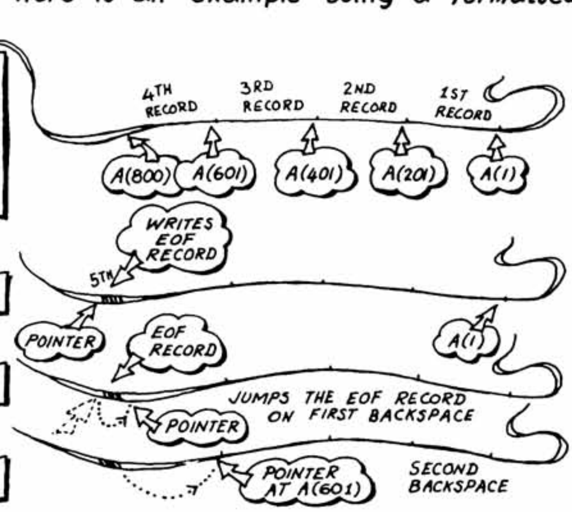
Because of the different behaviour of different Fortrans it is best not to use *BACKSPACE* with unformatted files. Here is an example using a *formatted* file.

```
      DATA    KFILE /8/
      DØ 10   K = 1, 601, 200
      L = K + 200 - 1
   10 WRITE (KFILE,100) (A(I), I=K,L)
  100 FØRMAT  (200F10.2)

      ENDFILE    KFILE

      BACKSPACE   KFILE

      BACKSPACE   KFILE
```



The forms of the auxiliary input and output statements are as follows:

ENDFILE *unit*
REWIND *unit*
BACKSPACE *unit*

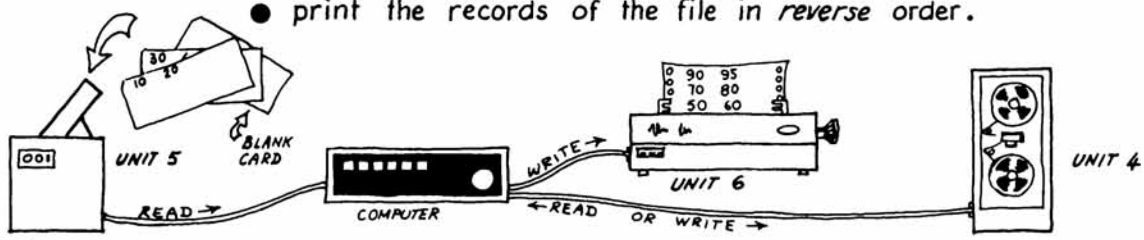
where:

unit is either an integer constant (digits only) or the name of an integer variable (not an array element). *unit* denotes a connection to a magnetic-tape deck or disk file. The connection is made by a command to the computer's operating system ~ perhaps by a *PROGRAM* command placed immediately before the main program.

These are *executable* statements and therefore may be labelled.

For the following example assume the installation shown. Units 5 and 6 are connected to the card reader and line printer according to the usual convention. Unit 4 is connected to a magnetic-tape deck or disk file. The object is:

- to read the waiting cards to a file on magnetic tape or disk
- print the records of the file in *reverse* order.



The end of the card deck is signified by a blank card. All other cards have non-zero integers right justified to columns 10 and 20.

```
      INTEGER READER, PRINTR, TAPDEK, NUMBER, I, M, N
C
      DATA READER, PRINTR, TAPDEK / 5, 6, 4 /
C
      REWIND  TAPDEK
      NUMBER = -1                                DON'T COUNT THE BLANK CARD
   10 NUMBER = NUMBER + 1
      READ (READER, 100)  N, M
      IF((N .EQ. 0) .AND. (M .EQ. 0)) GØ TØ  20
      WRITE (TAPDEK, 100)  N, M
      GØ TØ  10
C
   20 IF ( NUMBER .EQ. 0)  STØP  1
      ENDFILE   TAPDEK                           NO RECORDS
         DØ  30  I = 1, NUMBER
         BACKSPACE   TAPDEK                      FIRST TIME ROUND
         BACKSPACE   TAPDEK                      THIS SPACES OVER
         READ  (TAPDEK, 100) N, M                THE EOF RECORD
         WRITE (PRINTR, 200) N, M
   30    CØNTINUE
C
      REWIND  TAPDEK
      STØP
  100 FØRMAT   ( 2I10)
  200 FØRMAT   ( 1X,  2I10)
C
      END
```

EXERCISES

CHAPTER 11

11.1 Develop a program for writing card images (either from punched cards or from lines of data typed at a VDU) to a file on magnetic tape or disk. Do this for writing both formatted and unformatted files and compare the sizes of files created from the same card images. Most, if not all, installations allow users to discover the sizes of files they have created.

11.2 Develop a program for printing the contents of a magnetic-tape or disk file of card images. Your Fortran probably provides a means of detecting the EOF record. Use the facility provided but do discover whether or not it conforms to the Fortran 77 standard.

11.3 Amend one or two of your favourite programs to read from magnetic-tape or disk file – and write results to another magnetic-tape or disk file. Use the programs specified in 11.1 and 11.2 above to provide the data and print results.

12

MORE WORKED EXAMPLES

LINEAR SIMULTANEOUS EQUATIONS
SHORTEST ROUTE THROUGH A NETWORK
REVERSE POLISH NOTATION
EXERCISES

LINEAR SIMULTANEOUS EQUATIONS

Computers spend many hours solving sets of linear simultaneous equations. They crop up in engineering (stress analysis of bridges, buildings, aircraft) and many other fields. The solution of a set of several thousand equations takes much ingenuity to devise because intermediate results have to be filed on disk and rapidly recalled. And there is the problem of accuracy; solution in a badly chosen order can yield useless answers or none at all. The example here illustrates just the bare bones of the problem.

Consider these three simultaneous equations (called *linear* because there is no x^2, y^3 *etc.*):

$$\begin{aligned} 15x + 10y + 5z &= 4.2 \\ 12x + 24y + 8z &= 7.3 \\ 6x \phantom{{}+24y} + 36z &= 3.5 \end{aligned}$$

Recall that we are allowed to multiply any equation all the way through by a non-zero multiplying factor; also to subtract one equation term by term from another so as to replace either of these equations. Neither of these operations changes the solution unless we start getting small differences between large quantities hence introducing inaccuracy. The *aim* of these operations (demonstrated below) is to create a new set of equations looking like this:

$$\begin{aligned} 15x + 10y + 5z &= 4.2 \\ 16y + 4z &= 3.94 \\ 35z &= 2.805 \end{aligned}$$

Once the equations are reduced to this *triangulated* form it is easy to solve them. Starting with z: $z = 2.805 \div 35 = 0.0801$.
Knowing the value of z substitute in the second equation to find y: $16y + 4 \times 0.0801 = 3.94$ hence $y = (3.94 - 4 \times 0.0801) \div 16 = 0.2262$. Knowing the values of y and z substitute in the first equation to find x: $15x + 10 \times 0.2262 + 5 \times 0.0801 = 4.2$ hence $x = (4.2 - 10 \times 0.2262 - 5 \times 0.0801) \div 15 = 0.1025$. In summary $x = 0.1025$, $y = 0.2262$, $z = 0.0801$. This is called *back substitution*. But first we must *triangulate*.

To get rid of the coefficient of x in the second equation subtract a multiple of the first equation term by term from the second ≈ and so create a new second equation. To do this the multiplying factor must obviously be $12 \div 15$:

2^{nd} *equation*:	$12x$	$+ 24y$	$+ 8z$	$=$	7.3
minus:	$(12 \div 15) \times 15x$	$+ (12 \div 15) \times 10y$	$+ (12 \div 15) \times 5z$	$=$	$(12 \div 15) \times 4.2$
new 2^{nd} *eq.*:	$0x$	$+ 16y$	$+ 4z$	$=$	3.94

To get rid of the coefficient of x in the third equation subtract a multiple of the first equation term by term from the third. This time the multiplying factor must be $6 \div 15$:

3^{rd} *equation*:	$6x$		$+ 36z$	$=$	3.5
minus:	$(6 \div 15) \times 15x$	$+ (6 \div 15) \times 10y$	$+ (6 \div 15) \times 5z$	$=$	$(6 \div 15) \times 4.2$
new 3^{rd} *eq.*:	$0x$	$- 4y$	$+ 34z$	$=$	1.82

We have now finished eliminating coefficients of x (and incidentally have finished using multiples of the first equation which remains unchanged). The equations now look like this:

$$\begin{aligned} 15x + 10y + 5z &= 4.2 \\ 16y + 4z &= 3.94 \\ -4y + 34z &= 1.82 \end{aligned}$$

To get rid of the coefficient of y in the third equation subtract a multiple of the second equation from the third. The factor is $-4 \div 16$.

3rd equation:	$-4y$	$+ 34z =$	1.82
minus :	$(-4 \div 16) \times 16y$	$+ (-4 \div 16) \times 4z =$	$(-4 \div 16) \times 3.94$
new 3rd eq. :	$0y$	$+ 35z =$	2.805

At last the equations are triangulated. Notice the second equation remains unchanged from above:

$$\begin{aligned} 15x + 10y + 5z &= 4.2 \\ 16y + 4z &= 3.94 \\ 35z &= 2.805 \end{aligned}$$

and the solution may be found by back substitution as already demonstrated.

This method works as long as you don't divide by zero when forming the multiplying factor ≈ and gives good results when coefficients *on* the diagonal (15, 24, 36 opposite) are large compared to those *off* the diagonal.

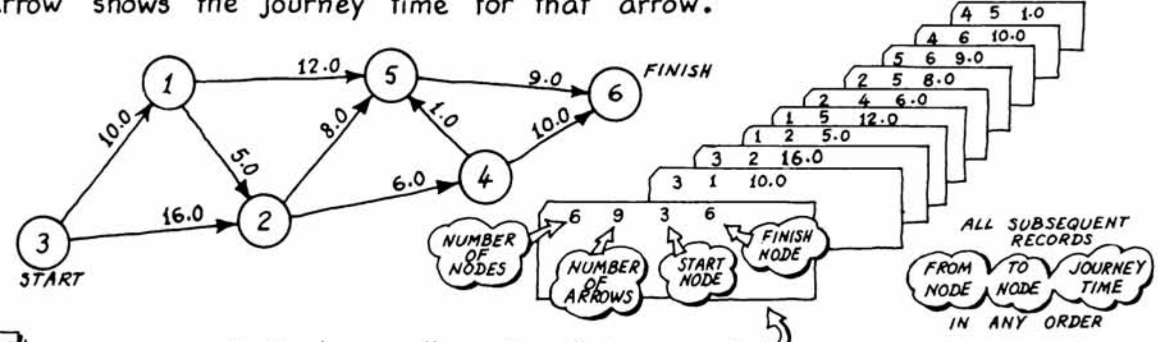
The following subroutine operates on array *A(,)* in which the numerical coefficients of the equations are stored ≈ and vector *B()* in which the right-hand side is stored. The third parameter, *NUMBER*, specifies the actual number of equations to be solved. The subroutine changes array *A(,)* to triangulated form and *replaces* the contents of vector *B()* with the values of the "unknowns".

```
      SUBROUTINE  EQUNS (A, B, NUMBER)
      INTEGER  NUMBER, PENULT, I, J, K, NEXT
      REAL  A(NUMBER, NUMBER), B(NUMBER), DIVISR, FACTOR
C                                      "ADJUSTABLE" DIMENSIONS
C ELIMINATE  COEFFICIENTS                   OF A(,) & B()
      PENULT = NUMBER - 1
        DO 20  I = 1, PENULT
        DIVISR = A(I,I)
        IF (ABS(DIVISR).LT. 0.000001) STOP 1      SEE EXERCISE 12.1
        NEXT = I + 1
          DO 30  J = NEXT, NUMBER
          FACTOR = A(J,I) / DIVISR
            DO 40  K = 1, NUMBER
            A(J,K) = A(J,K) - FACTOR*A(I,K)
   40       CONTINUE
          B(J) = B(J) - FACTOR*B(I)
   30     CONTINUE
   20   CONTINUE
C
C BACK  SUBSTITUTE
        DO 50  I = 1, NUMBER                      I COUNTS "UP"
        K = NUMBER - I + 1                        K COUNTS "DOWN"
        IF ( K  .EQ. NUMBER ) GO TO 70
        NEXT = K + 1
          DO 60  J = NEXT, NUMBER
          B(K) = B(K) - B(J)* A(K,J)               FIRST TIME ONLY
   60     CONTINUE
   70   B(K) = B(K)/ A(K,K)
   50   CONTINUE
C
      RETURN
C
      END
```

SHORTEST ROUTE

THROUGH A NETWORK
(ILLUSTRATING USE OF CHAINS)

Finding the shortest (or longest) route through a network is a problem that crops up in various disciplines ⇝ one of which is *critical path analysis* for the control and monitoring of construction projects. Given a network such as that below the problem is to find the shortest route from the node marked START to that marked FINISH. The journey must at all times be in the direction of the arrow. The number against each arrow shows the journey time for that arrow.



The program starts by reading the first record:

```
      INTEGER  NØDES, ARRØWS, START, FINISH, RØUTE(50),
     1         HEAD(50), LINK(120), TIP(120), KØUNT
      REAL     TIME(120), BESTIM(50), BETTER
      LØGICAL  SWITCH(50), ØN, ØFF
C
      DATA  ØN, ØFF / .TRUE. , .FALSE. /
C
      READ (5, 100) NØDES, ARRØWS, START, FINISH
  100 FØRMAT (4I5)
```

ALLOW 50 NODES AND 120 ARROWS

Then four entities are established for each node as illustrated below:

where the switch (explained later) is set to ON ; the best time to this node is set impossibly high except for the START node for which the best time is invariably zero. The head of a chain linking all nodes running *out* of this node is set to zero; so is the link on the chain to be created along the best route.

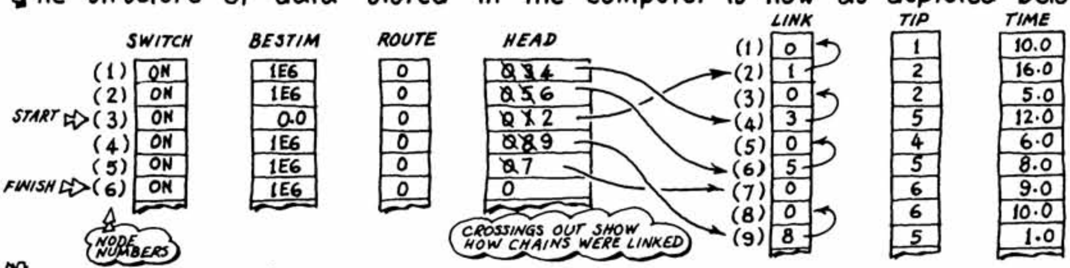
```
      DØ 10  I = 1, NØDES
      SWITCH (I) =  ØN
      BESTIM (I) =  1E6
      HEAD(I) = 0
      RØUTE(I) = 0
   10 CØNTINUE
      BESTIM(START) = 0
```

Now the remaining data are read. The node at the tip of each arrow is read into the vector named TIP( ) ; the journey time is read into the vector named TIME( ); the node at the tail of each arrow is linked into a chain held in the vector LINK( ) and with its head in vector HEAD( ).

```
      DØ 20   J = 1, ARRØWS
      READ (5, 200)  N,  TIP(J),  TIME(J)
      LINK (J)  =  HEAD(N)
      HEAD (N)  =  J
   20 CØNTINUE
  200 FØRMAT  (2I5, F10.0)
```

LINK INTO CHAIN

The structure of data stored in the computer is now as depicted below:



Now for the working part of the program. Start at node *START* and visit each node in turn: 3, 4, 5, 6,... going round again..., 1, 2, 3, 4,... until all switches are *off*.

```
      K = START
      GØ TØ  40          ⇨ Ⓐ
```

At each node consult the switch: if *off* move to the next node: if *on* then run down the chain of arrows out of this node. For each arrow add its journey time to the best time so far achieved at the tail ⇔ and see if this sum gives a better route to the tip. If so switch *on* at the arrow's tip and replace the previous best time with the new one. Also put the node number at the tail into the vector *ROUTE()* so as to build a chain along the best route. Having completed all this work at one node switch *off* that node. Keep variable *KOUNT* for counting the number of cycles through the nodes ⇔ but reset to zero every time you meet a node which is *on*. So when *KOUNT* reaches 2 all switches are *off*.

```
   30 K = K + 1
      IF  (K .LE. NØDES)  GØ TØ  35
      K = 1
      KØUNT  =  KØUNT + 1
      IF  (KØUNT  .GT.  1 )  GØ TØ  85     ⇨ Ⓑ   ALL SWITCHES ARE OFF
   35 IF  ( .NØT.  SWITCH(K) )  GØ  TØ  30
Ⓐ ⇨ 40 KØUNT  = 0
      N = HEAD (K)          ⇦ HEAD OF CHAIN FOR NODE K
   50 IF  (N .EQ. 0)  GØ TØ  80             END OF CHAIN
   60 BETTER  =  BESTIME (K)  + TIME(N)
      L = TIP(N)
      IF( BESTIM(L) .LT. BETTER)  GØ TØ  70
      SWITCH (L) =  ØN
      BESTIM (L) =  BETTER
      RØUTE (L)  =  K
   70 N = LINK(N)
      GØ TØ  50
   80 SWITCH (K)  =  ØFF
      GØ  TØ  30
```

When all switches are *off* the output may be printed:

```
Ⓑ ⇨ 85 WRITE  (6, 300) START, FINISH, BESTIM(FINISH)
  300 FØRMAT(16H BEST RØUTE FRØM, I4, 3H TØ, I4, 6H TAKES,
     1          F6.1,  8H THRØUGH )
      N  =  FINISH
   90 IF  (N .EQ. 0)   STØP
      WRITE  (6,400) N
      N  =  RØUTE (N)          ⇦ CLIMB DOWN ROUTE CHAIN
      GØ  TØ  90
C
  400 FØRMAT  (1X,  I4)
      END
```

Using the data opposite, this program should say the best route from 3 to 6 takes 31.0 (units of time) through nodes 6, 5, 4, 2, 1, 3.

REVERSE POLISH NOTATION

ILLUSTRATING USE OF STACKS

Algebraic expressions may be converted from the conventional form to a form without any parentheses. This notation is called *reverse Polish*. For example:

$A+(B-C)*D-F/(G+H)$ becomes $ABC-D*+FGH+/-$

The new expression is easier to evaluate than might appear. For example let $A=6.0$, $B=4.0$, $C=1.0$, $D=2.0$, $F=3.0$, $G=7.0$, $H=5.0$. With these values the expression to be evaluated is:

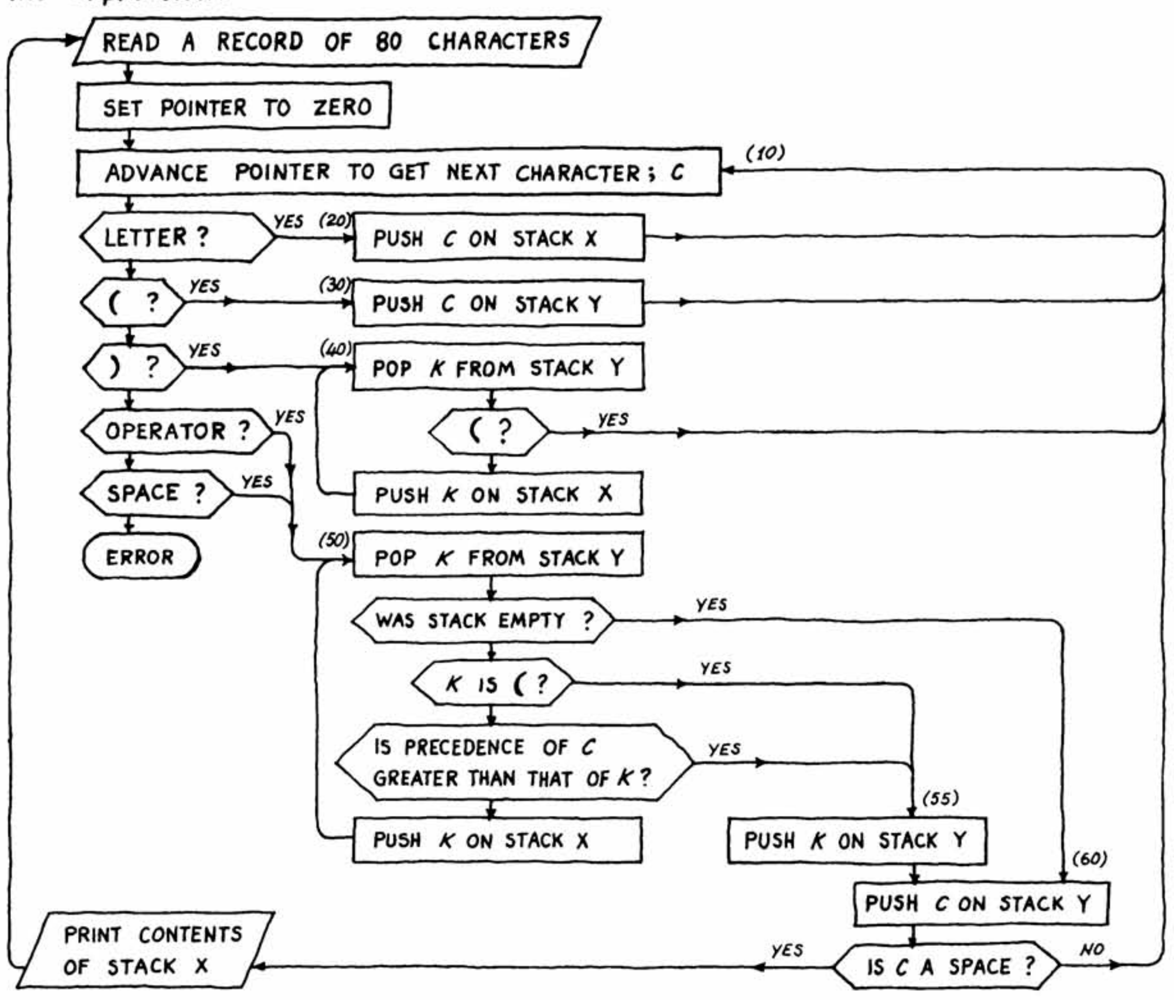
6.0 4.0 1.0 − 2.0 * + 3.0 7.0 5.0 + / −

Work from left to right pointing to each item in turn. Whenever you come to an operator *apply that operator to the previous pair of terms.* This procedure is illustrated below in steps:

```
6.0   4.0   1.0   −  ⇦ OPERATOR
6.0         3.0   2.0   *  ⇦ OPERATOR
6.0               6.0      +  ⇦ OPERATOR
                  12.0     3.0   7.0  5.0   +  ⇦ OPERATOR
                  12.0     3.0       12.0      /  ⇦ OPERATOR
                  12.0               0.25         −  ⇦ OPERATOR
                                     11.75  ⇦ RESULT
```

IN OTHER WORDS
6.0+(4.0−1.0)*2.0−3.0/(7.0+5.0)
= 11.75

The logic for converting an expression into reverse Polish notation is given below as a flow chart. Two stacks are used and referred to as STACK X and STACK Y. Operators have *precedence*: * and / have the highest (no exponentiation in this example) + and − come next; finally *space* is considered an operator with lowest precedence. A space is used to terminate the expression.



The following program assumes the existence of subroutines *PUSHX, POPX, PUSHY, POPY* for managing two integer stacks. These may be modelled on *PUSH(EXPRN)* and *POP(TOP, OK)* on page 79 but also declare *COMMON/STACKS/* as in the main program. The program also refers to a function subprogram named *PRECED(I)* for returning the precedence of an operator ≈ using values given to characters by function *INDEX(N)* on page 89:

```
      INTEGER FUNCTIØN  PRECED(I)
      INTEGER DIV, MULT, MINUS, PLUS, SPACE
      DATA  DIV, MULT, MINUS, PLUS, SPACE / 42, 41, 40, 39, 37 /
      IF ((I .EQ. DIV) .ØR. (I .EQ. MULT)) PRECED = 3
      IF ((I .EQ. PLUS) .ØR. (I .EQ. MINUS )) PRECED = 2
      IF ( I .EQ. SPACE) PRECED = 1
      RETURN
      END
```

Here is the program itself: it reads a record containing an expression composed of letters, operators and brackets ≈ then prints the expression in reverse Polish notation. The program returns to read and decode a new expression starting in column 1 but stops if it meets a dollar sign.

```
      LØGICAL  ØK
      INTEGER CARD(80), PØINTR,C,K, LEFT, RIGHT, SPACE, PLUS,MINUS,
     1        MULT, DIV, DØLLAR, A, Z, PRECED
      CØMMØN /SIGMA/ ICHR(47)
      CØMMØN / STACKS / ISX(20), IPX, ISY(20), IPY
      DATA  LEFT, RIGHT, SPACE, PLUS, MINUS, MULT, DIV, DØLLAR, A, Z
     1  /   43,   44,   37,   39,    40,   41,   42,    47,11,36/
C
    5 READ (5, 100) CARD
  100 FØRMAT  (80A1)
      PØINTR = 0
   10 PØINTR = PØINTR + 1
      C = INDEX ( CARD (PØINTR))
      IF (C .EQ. DØLLAR)  STØP
      IF ((C.GE. A) .AND. (C .LE. Z)) GØ TØ 20
      IF ( C .EQ. LEFT)  GØ TØ 30
      IF ( C .EQ. RIGHT)  GØ TØ 40
      IF ((C .EQ. PLUS) .ØR. (C.EQ. MINUS))GØ TØ 50
      IF (( C.EQ. MULT) .ØR. ( C .EQ. DIV)) GØ TØ 50
      IF (C .EQ. SPACE) GØ TØ 50
      STØP 1
C
   20 CALL  PUSHX(C)
      GØ TØ 10
   30 CALL  PUSHY(C)
      GØ TØ 10
   40 CALL  PØPY (K, ØK)
      IF (.NØT. ØK)  STØP 2
      IF ( K .EQ. LEFT)  GØ TØ 10
      CALL  PUSHX(K)
      GØ TØ 40
   50 CALL  PØPY (K, ØK)
      IF (.NØT. ØK)  GØ TØ 60
      IF ( K .EQ. LEFT)  GØ TØ 55
      IF (PRECED(C) .GT. PRECED(K)) GØ TØ 55
      CALL  PUSHX (K)
      GØ TØ 50
   55 CALL PUSHY (K)
   60 CALL PUSHY (C)
      IF ( C .NE. SPACE) GØ TØ 10
C
    1 CALL  PØPX (K, ØK)
      IF ( .NØT. ØK)  GØ TØ 5
      WRITE (6, 200)  ICHR(K)
      GØ TØ 1
  200 FØRMAT (1X, A1)
C
      END
```

IPX & IPY SHOULD BE SET TO ZERO IN THE BLOCK DATA SUBPROGRAM

EXPRESSION STARTS IN COL. 1

A+(B−C)*D−F/(G+H)

STOPPING CONVENTION

$

LETTER

BRACKETS

ERROR: EMPTY STACK

ERROR

STACK WAS EMPTY

PRINT STACK X

−
/
+
H
G
F
+
*
D
−
C
B
A

EXERCISES

12.1 The example on simultaneous equations is deficient because it cannot cope with zero divisors. Assuming the equations do have a solution this deficiency might be overcome by solving the equations in a different order. Amend the example so that if there is a zero divisor the program looks down the remaining equations to find one with a non-zero divisor. Exchange this equation with the one giving trouble and proceed as before.

12.2 The example on simultaneous equations deals with just one right-hand side but could easily be made to deal with several ⇝ either in sequence or simultaneously. In particular, if the right-hand sides look like this:

$$\begin{matrix} 1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & \vdots & \vdots & \end{matrix}$$

AS MANY RIGHT-HAND SIDES AS THERE ARE EQUATIONS

then the array of resulting solutions is called the *inverse* of the matrix of coefficients. If you are familiar with matrix algebra write a subroutine to generate an inverse matrix. See Exercise 7.5.

12.3 Improve the shortest route program so that it makes some checks on the validity of input data. For example ensure there is no more than one arrow between any two nodes; at least one arrow pointing out of the *start* node; at least one pointing into the *finish* node. Provide error messages to be printed when such checks fail.

12.4 Change the program on reverse Polish notation so that it reads real numbers rather than letters. For example instead of reading:

*A + (B−C) * D − F / (G + H)*

change STACK X to type *REAL* and make the program capable of reading:

*6.5 + (3.76 − 2) * 7.9 − 13.3 / (3.14 + 2.8)*

To do this you may find it convenient to modify and use the free format input routine on page 115. Make it read unsigned numbers and terminate on meeting a space, operator or bracket.

12.5 Modify the above program so that it has the form of a subroutine:

SUBROUTINE READER (V, OK)

where *V* returns the next item to be read; logical variable *OK* reports the success or failure of any call to the subroutine.

Change the logic dealing with stacks. When an operator is about to be pushed on STACK X :

- pop two numbers from STACK X
- apply the operator that was about to be pushed
- push the resulting value on STACK X

When one complete expression has been dealt with there should be just *one* value on STACK X. This value should be returned by parameter V of subroutine *READER (V, OK)*.

This subroutine permits anyone who writes numerical data for your programs (those employing *READER*) to include *expressions* instead of numbers wherever he or she chooses.

```
2*3.14*27.6     3.9*(1.1-0.03)/(6*(7.25+4))
```

WOW!

BIBLIOGRAPHY

The Fortran described in this book (Fortran 66) is formally defined in:

AMERICAN STANDARD FORTRAN
ANSI X3.9-1966
American National Standards Institute Inc.
New York, N.Y., U.S.A

An amplified description of Fortran 66 including many examples of syntax may be found in:

STANDARD FORTRAN PROGRAMMING MANUAL
SECOND EDITION 1972
National Computing Centre Limited
Manchester, U.K.
ISBN 0 85012 063 2

A commentary on the above — defining a subset of Fortran 66 for writing portable programs — has been made. This commentary is the fruit of many years experience in making large programs run on a range of different computers in Britain and abroad:

HECB PROGRAMMING INSTRUCTION MANUAL
VOLUME III - FORTRAN 1979
Highway Engineering Computer Branch
Department of Transport, St Christopher House,
Southwark Street, London SE1 , U.K.

Another study of Fortran for writing portable programs — again drawing upon much experience — has been published:

COMPATIBLE FORTRAN
A. Colin Day, 1978
Cambridge University Press
ISBN 0 521 22027 0

The full Fortran 77 language is formally defined in:

AMERICAN NATIONAL STANDARD PROGRAMMING LANGUAGE FORTRAN
ANSI X3.9 - 1978
American National Standards Institute Inc.
New York, N.Y., U.S.A

and it is well to check whether the facilities offered by the Fortran you are using — and which fall outside the scope of Fortran 66 — conform to this definition. This particularly concerns the use of files; an area in which Fortran 66 is deficient and different Fortrans employ different solutions.

Many useful tricks of the programmer's art are beautifully described in:

FORTRAN TECHNIQUES (WITH SPECIAL REFERENCE TO NON-NUMERICAL APPLICATIONS)
A. Colin Day, 1972
Cambridge University Press
ISBN 0 521 08549 7 & 0 521 09713 3 (paperback)

INDEX

PRINCIPAL REFERENCES, WHERE SUCH EXIST, ARE PUT FIRST IN THE LIST

INDEX (CONTINUED)

R

S

T

U

V

W

Z

Fortran is a computer language that has been around for a quarter of a century, and is still much used despite predictions throughout its life that it would be replaced by more elegant languages. But Fortran is still the only language in which it is possible ≈ with care ≈ to write truly portable programs.

Using his own special formula ≈ original and readable style together with unique calligraphy ≈ Donald Alcock now does for Fortran what he did for BASIC in his earlier book, *Illustrating BASIC*. The Fortran described and illustrated is that defined by the American National Standards Institute in 1966 (Fortran 66) but with allusions to the same Institute's standard published in 1978 (Fortran 77).

For both the person who is new to computers and programming, and the person who has met only BASIC before, this book will give an excellent introduction to standard Fortran; it also provides a reference manual for the language ≈ one that emphasizes the self-discipline needed to write portable programs. The book also illustrates a few useful tricks of the programmer's trade.

Also by the author

ILLUSTRATING BASIC

'Alcock takes the reader through the elements of the language, pausing now and again to pass on useful tips about list processing, matrix algebra, symbol state tables and network analysis ≈ indeed, for these tips alone, the book makes worthwhile reading for any programmer whether he is familiar with Basic or not.'

Computer Bulletin

'Humane, interesting, comprehensive, and ≈ in paperback form, at least ≈ excellent value for money. Congratulations to the author. Very highly recommended.'

Practical Computing

CAMBRIDGE UNIVERSITY PRESS

0 521 28810 X